Electronic Design Automation Report

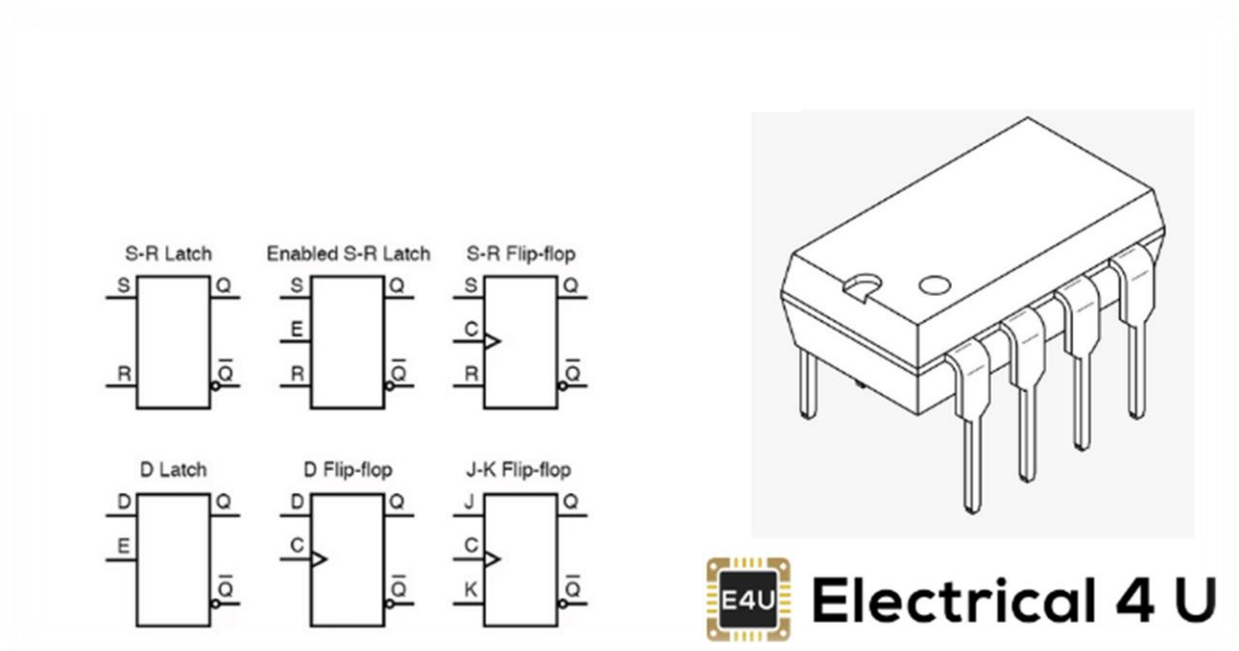
1. Introduction

A switch is a systems administration module that cradles and advances information (in packets design) across an organization toward their objections, through a cycle known as directing. Switches use headers and sending tables to decide the best way for sending the packets as indicated by plan prerequisites, and they use correspondence conventions to speak with one another and conjure the best course between any two hubs. Switches are not just utilized in PC organizations applications, yet additionally have been coordinated in framework on-Chip (SoC) based designs to frame what is know as Organizations on-Chip (NoC) applications. With SoC plans that have many Protected innovation (IP) cores1, interconnection utilizing a solitary shared transport isn't adequate any longer. The NoC is another worldview that gives an incorporated answer for accomplishing effective between module correspondence. Dissimilar to PC organizations, NoC has more limited correspondence delay furthermore, restricted silicon territory which put more impediments on the NoC-based framework plan and require various ways to deal with beat configuration challenges uncommonly in switch plan,When packets show up at the contribution of the switch, they are cushioned at the information support, at that point the regulator peruses the bundle header and congress the change texture to coordinate the bundle to the proper yield cradle. The yield support is a gathering of FIFOs intended to handel various access demands at the yield ports. Directing tables also, timing synchronization are done in the control part. We used VHDL language in the design.

2. Design Flow

- Integrated Circuits (IC)

Also known as **microelectronic circuit,** Chip or microchip, a single-unit assembly of electronic components in which miniaturized active devices (e.g., transistors and diodes) and passive devices (e.g., capacitors and resistors) and their interconnections are constructed on a thin semiconductor material substrate (typically silicon). Therefore, the resulting circuit is a tiny monolithic "chip" that may be as small as a few square centimeters or a few square millimeters. In general, the individual circuit components are microscopic in size.



Transistors are the main components of an IC. Based on the applications of ICs, these transistors can be bipolar or field effect.

- Process flow

The process of developing an IC design to the point at which it is possible to produce the IC in a semiconductor manufacturing plant (i.e., a foundry). In order to record, simulate, optimize, and identify errors during the process, this includes the use of sophisticated system and process models as well as mathematical tools and software.

IC System Design

Assuming your IC specifications are completed and approved by the different parties, it's time to start thinking about the architectural design. In IC system design phase, the entire IC functionality is broken down to small pieces with clear understanding about the block implementation. For example: for an encryption block, do you use a CPU or a state machine. Some other large blocks need to be divided into subsystems and the relationship between the various blocks has to be defined. In this phase the working environment is documentation.

Register Transfer Level (RTL)

For digital ICs or for digital blocks within a mixed-signal IC, this phase is basically the detailed logic implementation of the entire IC. This is where the detailed system specifications is converted into VHDL or Verilog language. In addition to the digital implementation, a functional verification is performed to ensure the RTL design is done according to the specifications.
When all the blocks are implemented and verified the RTL is then converted into a gate level netlist.

Synthesis

In this phase the hardware description (RTL) is converted to a gate level netlist. This process is performed by a synthesis tool that takes a standard cell library, constraints and the RTL code and produces a gate-level netlist.
Synthesis tools are running different implementations to provide best gate level netlist that meets the constraints. It takes into account power, speed, size and therefore the results can vary much from each other. To verify whether the synthesis tool has correctly generated the gate-level netlist a verification should be done.

Layout

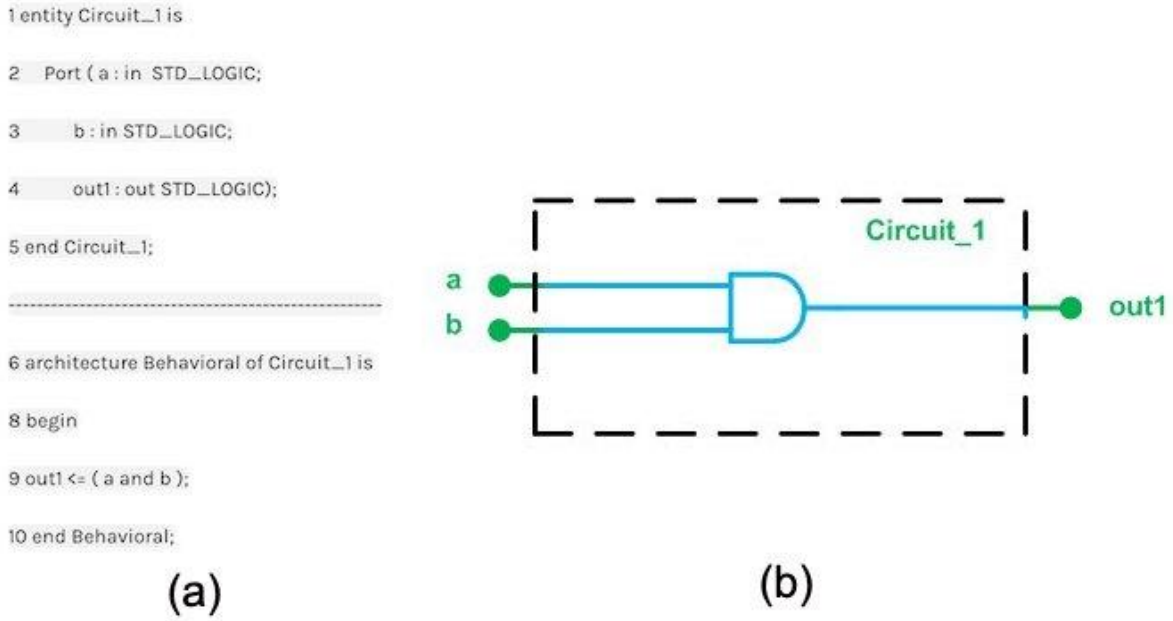
In this stage, the IC gate level netlist is converted to a complete physical geometric representation. The first step is IC floor planning which is a process of placing the various blocks and the I/O pads across the IC area based on the design constraints. Then placement of physical elements within each block and integration of analog blocks or external IP cores is performed. When all the elements are placed, a global and detailed routing is running to connect all the elements together.
Also, after this phase a complete simulation is required to ensure the layout phase is properly done.

➢ Digital IC design

A procedural process involving the conversion into digital blocks of specifications and features and then further into logic circuits. Many of the limitations associated with digital IC design derive from the process of casting and technological constraints.

- **Synthesis and Verification: Hardware Description Language and Functional Verification**

  It is important to convert the digital blocks with behavior descriptions developed in the early stages of digital design into a hardware description language (HDL), such as Verilog or VHDL. This stage is also referred to as the Register Transfer Level (RTL) step, which typically involves functional verification to ensure that the implementation of logic meets high-level specifications.



(a) An example of HDL code and (b) the circuit it describes

The hardware description is then converted into a gate-level netlist after this step, during which a variety of implementations and optimization routines can be attempted to better meet design objectives. At this stage, important considerations include power budget, velocity, footprint, and reliability.

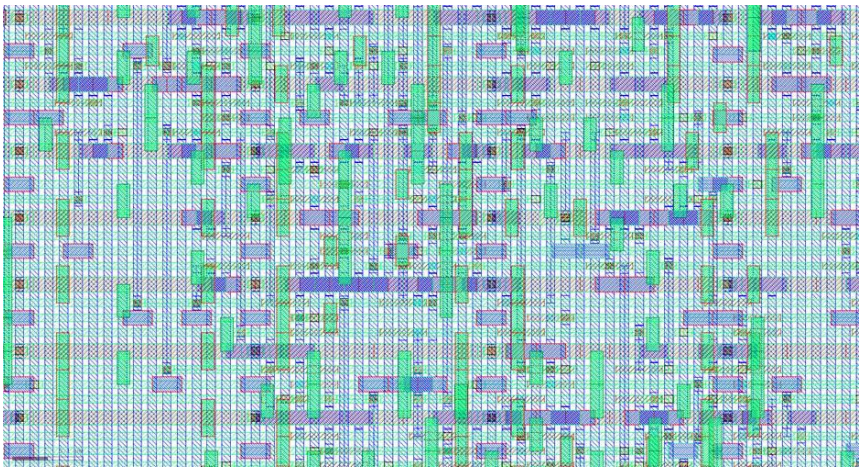
Physical IC Layout: Floorplanning and IP Cores

The gate-level netlist is converted into a physical layout following synthesis and verification, which is a geometric representation of the IC's layers and physical structure. Floorplanning techniques are used to ensure that the blocks and pads are placed throughout the IC to meet design objectives.

Components of the digital IC layout are often done using scripts and automated software processes due to the structured and repetitive nature of some digital blocks, such as memory and registers. During this stage, external IP cores are also placed, where the software reveals only the necessary interface portions of the IP. Routing automation scripts and software are used to connect each element after all blocks and gates are placed, along with manual routing, if necessary.

Verification and Simulation: Tapeout and Testing

Verification and simulation are then carried out, both of which must take into account the placement and the layout's physical features. The result, if successful, is an output file, such as GDSII (GDS2), which the foundry uses to make the ICs, the tape-out stage, with internal software and processes. In some instances,



The *layout of a chip* after place and route. Image used courtesy of Cadence Design Systems.

A small batch of first-run or prototype ICs are produced after tape-out, so that testing can be performed. Depending on the performance and economics of producing the IC, this testing may result in redesign or process changes.

➢ Design Flow

- Design Specification
  - Specifications
  - Constraints
  - Test bench development

- High-level system design
  - ➢ Design Partition
  - ➢ Entry-Verilog Behavior Modeling
  - ➢ Simulation/Functional Verification
  - ➢ Integration & Verification

- Logic Synthesis
  - Register Transfer Level (RTL) conversion into netlist
  - Design partitioning into physical blocks
  - Timing margin and timing constraints
  - RTL and gate level netlist verification
  - Static timing analysis

- Floorplanning
  - Hierarchical IC blocks placement
  - Power and clock planning

- Physical Synthesis

  Physical Synthesis is where the synthesis flow uses its knowledge of the target device's physical layout and timing to achieve the minimum use of the region at the necessary speed.
  - Timing constraints and optimization
  - CTS to minimize the skew and latency.
  - Routing create physical connections based on logical connectivity.
  - Static timing analysis
  - Update placement
  - Update power and clock planning

- Block Level Layout
  - Complete placement and routing of blocks

- IC Level Layout
  - IC integration of all blocks
  - Cell placement
  - Scan chain/clock tree insertion
  - Cell routing
  - Physical and electrical design rules check (DRC)
  - Layout versus schematic (LVS)
  - Parasitic Extraction
  - Post-layout timing verification
  - GDSII creation
  - Tape-out

- Examples of the CAD tools that could be used to complete the chip design flow.
  - LASI is a PC-based CAD program used for design of the physical layout of IC.
  - VASY (VHDL Analyzer for Synthesis). Converts a file written in VHDL VASY subset of the subset of VHDL Alliance.
  - BOOM (Boolean Minimization). ...
  - BOOG (Binding and Optimizing on Gates). ...
  - LOON (Local Optimizations of Nets). ...
  - ASIMUT (A Simulation Tool for hardware description).

3. Literature Review

To help plan efficiency it is pivotal that the push to add new parts to a given plan doesn't rely upon the size of the current plan as it may, just on the size of the new parts. All in all, the plan exertion should be a straight capacity of the size of the new parts. In the event that this is the case, huge parts and squares of past plans can be reused and the plan exertion can be put into the new parts. This is additionally an essential to give a strong technique, design, and along these lines a stage, that are manageable more than a few innovation ages. The focal postulation of this section is that an Organization on-Chip (NoC) has the possibility to give a particularly supportable stage and, if fruitful, will bring about a particularly critical change on the framework on-chip engineering what's more, plan measure that it tends to be known as a worldview change. On the other hand, on the off chance that it neglects to do as such, NoC will be only one of a few designs what's more, stages accessible to inserted framework fashioners.

A DESIGN METHODOLOGY FOR NOC-BASED SYSTEMS:

Abuse of a billion-semiconductor limit of a solitary ASIC requires new framework ideal models and critical upgrades to plan productivity.The Organization on-Chip sorts of approaches introduced in before sections attempt to separate and overcome the utilization of accessible silicon surface. Mix of tens or then again many PCs into a solitary chip along with

offbeat message passing organization segments the plan into more sensible units
what's more, permits to advance the subsystems as indicated by application necessities . Such adaptable and particular structures are common strides in framework design advancement particularly when actual usage are thought of. Primary unpredictability and useful variety of such frameworks are the challenges for the plan groups. Straightforward extrapolations of plan efficiency guides uncover that all around soon just the biggest organizations can finish their ASIC projects in the event that the full silicon limit is utilized. In the more drawn out period, plainly plan limit will be the restricting variable for the framework unpredictability. Underlying unpredictability can be expanded by having more beneficial plan strategies and by placing more assets in plan work. Plan strategy improvement utilizing higher deliberation levels prompts the more convoluted also, costly plan space investigation and amalgamation undertakings. When expanding the quantity of creators, the correspondence overheads devour practically all the advantages. The most appealing methodology is the reuse of prior plans, since, supposing that the plan is reused at any rate multiple times the expense of making plan reusable is advocated. In stage based plan, the reuse is reached out to designs and ongoing reconfigurable stages copy normal structures of computational units. Expanding the number of doors can be and will be utilized for expanding the usefulness of the items. The billion-semiconductor limit on silicon and activity recurrence at gigahertz range implies that we will have tera operations every second limit with respect to applications. Notwithstanding programming furthermore, PC equipment, a similar framework can have DSP capacities, simple also, radio recurrence capacities, progressed controlling capacities, fake knowledge, and so forth The framework level choices should consider all the various requirements of utilization types. It will require basic language,ideas and phrasing. The improvement of the usage of applications will mean the joining of an assortment of memory innovations, simple gadgets, reconfigurable innovations and PC advancements into the same stage. The primary unpredictability will compel us to work at the framework level when creating NOC-based frameworks. It won't be conceivable to go into

subtleties when settling on choices concerning reuse or application mappings, for model. It will likewise compel us to create stage kind of structures and assets. We need to supplant the objective item necessities with additional theoretical ideas. We need to utilize the necessities of the item region.

Software and Application Interfaces:

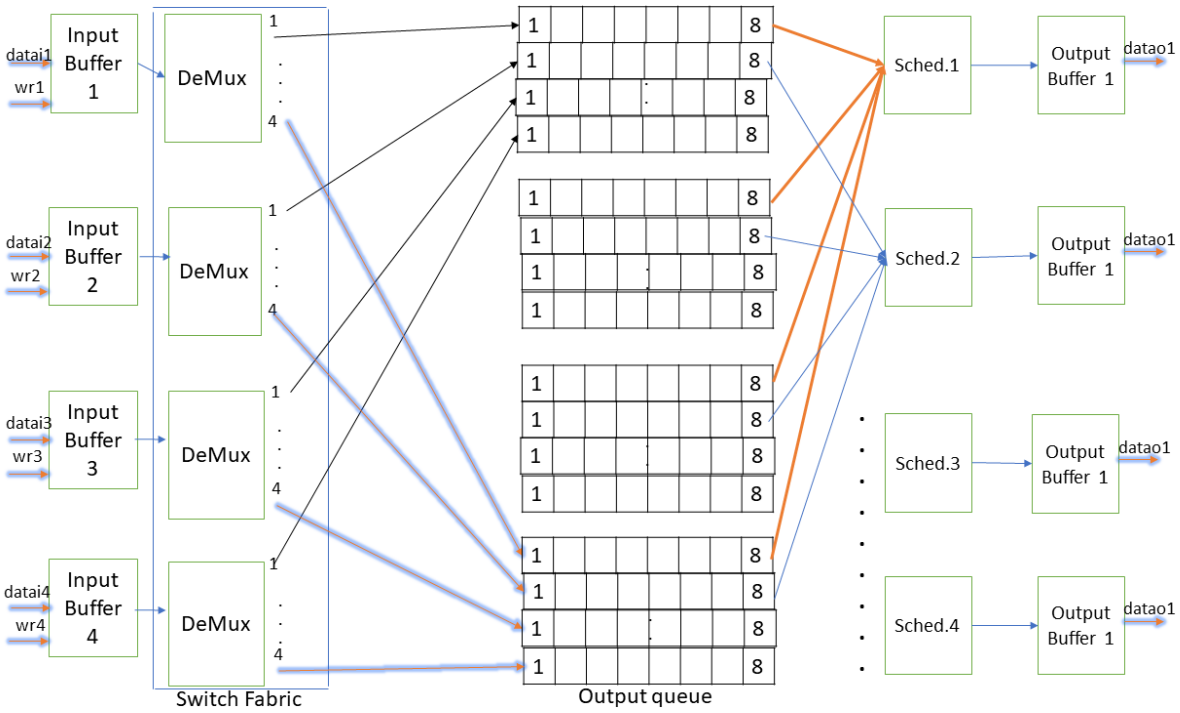
riginators will be able to coordinate more than fifty IP centers, every one of the size of a little processor, alongside numerous memory squares of various sorts on a solitary chip. To stay away from synchronization furthermore, clock slant issues, these multi-center SoCs will work utilizing Worldwide Offbeat Locally Coordinated (Ladies) worldview, where each center will work in a different clock area. The significant test the SoC specialists face today is to thought of organized, versatile, reusable and high execution interconnection structures/stages for incorporating an enormous number of heterogeneous centers on a solitary chip.

An interconnection stage is organized, if the hidden interconnection diagram has pleasant properties for its format and for numerical investigation. An adaptable design, in this unique situation, infers that there is no characteristic specialized constraint in the design to fabricate framework with huge number of centers. Reusability in all parts of framework configuration is critical to quick time to plan and trial of perplexing and huge SoCs. A reusable interconnection stage will necessitate that the architect just focuses on determination and specialization of centers and another SoC is arranged by connecting the centers in a pre-planned interconnection organization. The interconnection engineering should give offices to countless centers to proficiently trade data. This infers that the interconnection engineering ought to permit high correspondence data transmission between sets of centers just as simultaneous correspondence among numerous sets of centers. Numerous industrially significant applications, particularly in the region of multi-media correspondence and handling, require multi-center SoCs with high correspondence transfer speed among centers. Thusly, high execution in this setting suggests high correspondence transfer speed among centers.

Every one of these properties are connected however one property in engineering does not infer any remaining. Indeed, it is hard to guarantee all these attractive properties in an interconnection network all the while.

NoC is naturally a heterogeneous appropriated framework. Heterogeneity infers that various components, similar to assets, switches and interfaces, are planned in different methods. Various dialects, amalgamation apparatuses, programming compilers and linkers are needed for the plan of person components. There is no single plan stream which can be applied to the plan of every one of these components. Circulation suggests that measures on various assets interface with one another through the on-chip correspondence network. NoC configuration will be correspondence driven. Notwithstanding the NoC engineering, we likewise need to address the plan of cycle interchanges. On the off chance that we treat NoC as individual components, the plan of NoC correspondences might be convoluted. Correspondence principles also, conventions are wanted to organize programming NoC correspondences. The generally free plan of cycles makes it hard to incorporate every one of these components. For instance, how to program a cycle running on an ARM microchip to speak with a cycle running on an ASIC through the correspondence organization? To conceal the intricacy, we propose a NoC software engineer model where we offer interfaces to NoC creator.

4. Design Implementation



Module	VHDL Implementation & Function	Design challenges
Register	In the register, if the clock enable=1 and reset=0 then data would be transferred from input port to output port at the clock positive edge and if reset=1 then the output port is set to “00000000”	• Declaring entities with registered words like register. • Writing lines without a semicolon at the end.

Demux	If enable=1 then data will be transferred from input port to the suitable output port according to the selector value if selector=”00” then the output of port 1 will be equal to data input and if selector=”01” then the output of port 2 will be equal to data input and if selector=”10” then the output of port 3 will be equal to data input and if selector=”11” then the output of port 4 is equal to data input and if enable=0 then all the output ports will keep their current value.	• Writing lines without a semicolon at the end. • Forgetting declaring a certain library.
RAM	Ram function Its can give the access to write and read spontinously As its in the code source In begin if CLKA event anad CLKA is equals 1 and WEA equals 1 also so in that case the word written by user it is in write function In the second case As its in the code source In begin if CLKB event anad CLKB is equals 1 and REA equals 1 also so in that case the word written by user it is in read function	No challenges faced
Gray counter	The gray counter module is simply generating gray code representation ascendingly.	To study gray representation to implement it in combinational logic gates.
Gray to binary counter	This module converts gray code to binary code. From the study of both code representations and that it’s a 3-bit module, bit(2) in binary is the bit(2) gray bit; bit(1) is the xor of bit(2) bit and bit(1) in gray; bit(0) in binary is the xor of bit(2) bit, bit(1), bit(0) in gray.	To know the relation between gray and binary to form a combinational logic gates to implement the module
Controller	The controller module uses gray counter module and gray to binary module in order to specify where to write and read by making two pointers.	1.How to indicate full and empty signals? 2.How many processes to make the module? To do it in one process or more?

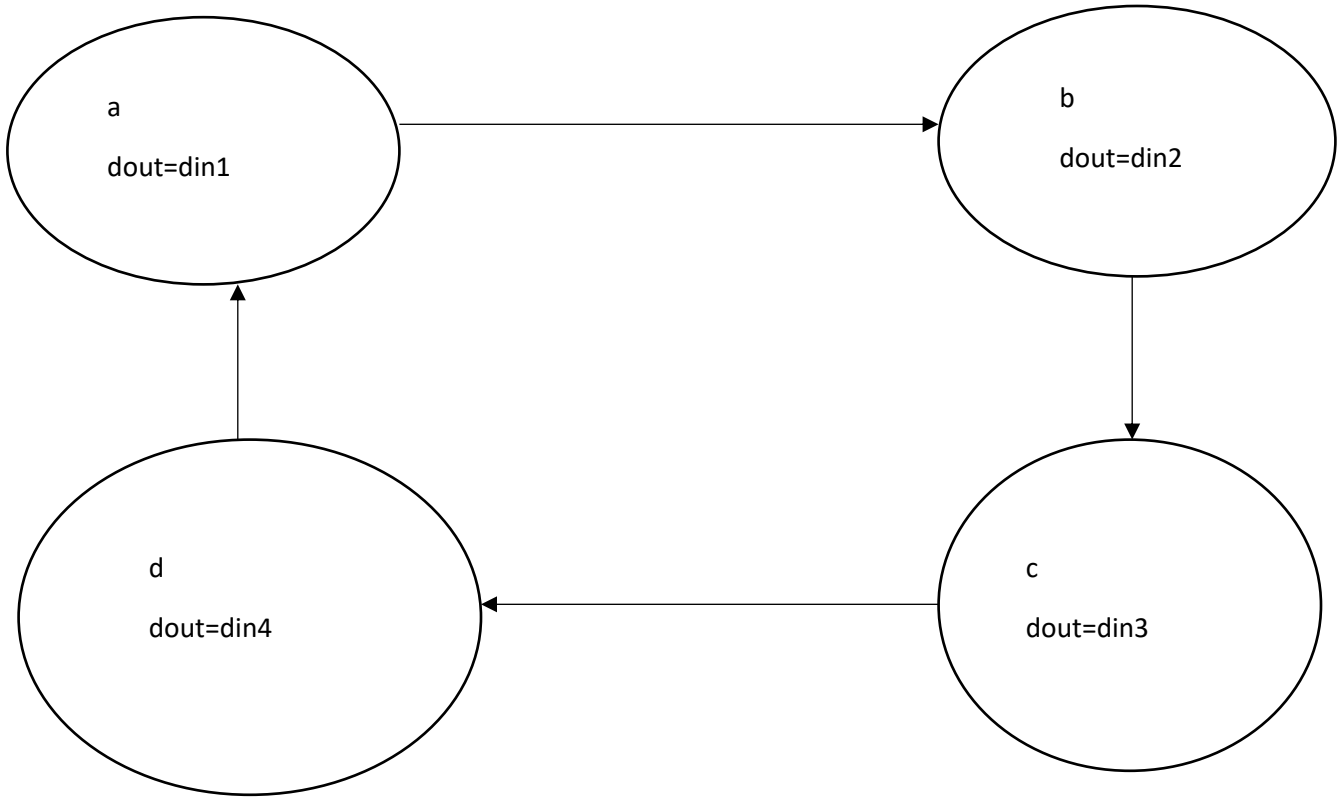
	First pointer is wr_ptr and second one is rd_ptr. Write pointer function only at write clock and that write valid is high and reset is low. Writing is valid when : write request and write clock are high and full signal is low, other than that writing is not functional. Same to reading, read pointer function only at read clock and that read valid is high and reset is low. Reading is valid when : read request and read clock are high and empty signal is low, other than that reading is not functional. We estimate empty and full flags using a counter signal to the fifo (fifo_counter) that counts how many packet data stored in the fifo and it's maximum value is 8. So, when reset is low, if (fifo_counter) is low then empty flag high, and if (fifo_counter) equals 8 then full flag is high. (fifo_counter) is incremented when writing and decremented when reading. This module was responsible for specifying where to read and where to write using pointers, it's condition (empty or full) and the validity of reading and writing. This module and ram module are going to be used in module 7 and be mapped to make the FIFO controller.	
FIFO Controller	In this module, we map module 3,6 in order to produce a full functional FIFO to use RAM as storage to read from or write in. By mapping module 6 and extracting pointers, read/write valid and empty/full in signals, RAM is then mapped to use these signals (read/write pointers and valid signals) in order to read the data from RAM which is the	1.Which signals, inputs and outputs should be mapped in RAM and Controller module.

	output of FIFO or write in RAM the data input that entered FIFO. Also FIFO takes full/empty signals as outputs.	
Round Robin	In this module, we have 4 input ports and one output port. In the first clock cycle data from input port 1 will be transferred to the output port and In the second clock cycle data from input port 2 will be transferred to the output port and In the third clock cycle data from input port 3 will be transferred to the output port and In the fourth clock cycle data from input port 4 will be transferred to the output port	• Writing lines without a semicolon at the end. • Forgetting declaring a certain library like IEEE.STD_LOGIC_1164.ALL library
4-port Router	This module has wr1,2,3,4 which are the request to write, and rd1,2,3,4 are the request to read as inputs. We map all previous modules to make the 4 port router. Simply, the router contains at first 4 registers that take the data packets to be written then the registers outputs go to demultiplexer inputs and according to the select signal inputs are then entering one of the 16 FIFO components, that is it in the writing process. In the reading process 4 Round robin schedulers loop around all the 16 FIFO components. Each schedular is holding 4 FIFO components to loop on and choose one each read clock cycle to then transfer it to the final 4 registers which their outputs are the routers outputs. But which FIFO components that are connected to each schedular? For example, first schedular is responsible for first FIFO component from each 4 consecutive FIFO component (FIFO 1,5,9,13) , second schedular is responsible for second FIFO	1.To use only 4 registers for reading and writing or 4 for reading and 4 for writing. 2.Schedular system 3. Carefully mapping tens of signals in the components.

	component from each 4 consecutive FIFO component (FIFO 2,6,10,14) and so on. Schedular is done like this so that when data is written from first input it will be read by first output, when data is written from second input it will be read by second output and so on. This system is clarified in the design diagram. Lastly in order to read a packet data, it should firstly written in high write clock then read it in the next clock (not same clock) at high read clock, so it at least takes two clock cycles to read a packet.	

5. Schedular Design and FSM Implementation

State Diagram of FSM For round robin Block



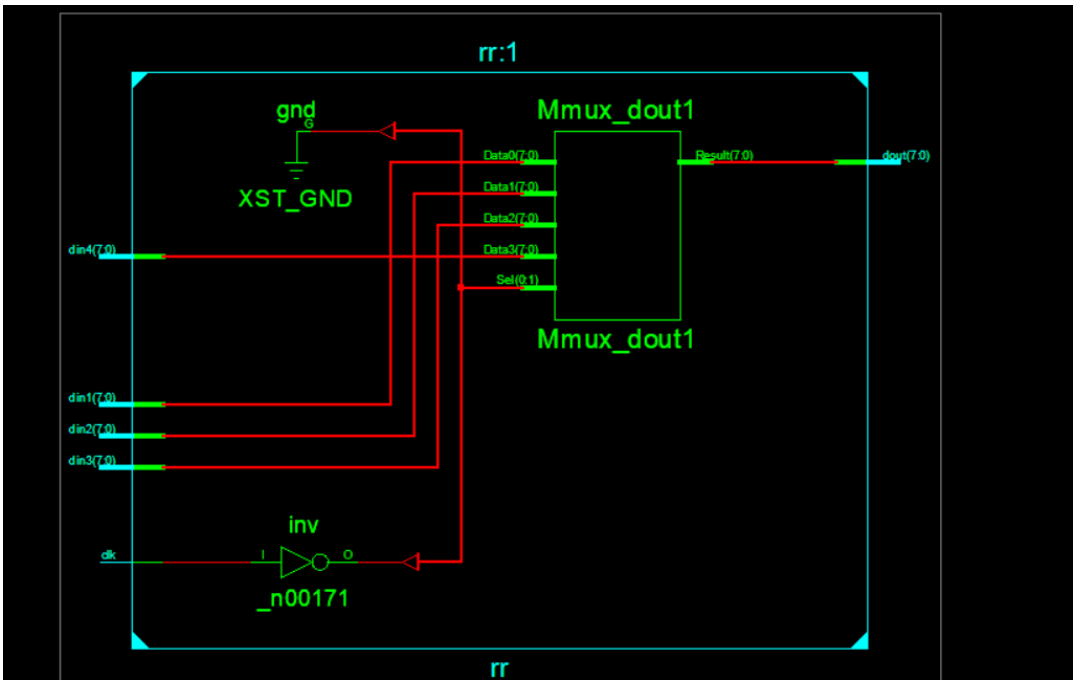
Process chosen

We chose Moore process as the output depends on the current state not the inputs.

A comparison between Single process, 2-process, 3-process

Points of comparison	Single process	2- process	3-process
Current state, next state, output,reset	Current state, next state, output,reset are implemented in one process	Current state and reset are implemented in the first process and next state and output are implemented in the second process	Current state and reset are implemented in the first process and next state is implemented in second process and the output is implemented in the third process

Synthesis Analysis



Timing Summary(analysis):

Speed Grade: -3

Minimum period: No path found
Minimum input arrival time before clock: No path found
Maximum output required time after clock: No path found
Maximum combinational path delay: 0.921ns

6. Test and Simulation Results

We at first reset to remove any garbage values.

First Write		First read		Reading and writing		Reset	
Datain		Datain		Datain		Datain	
1	00000001	1	00000000	1	00000001	1	00000001
2	00000001	2	00000000	2	00000001	2	00000001
3	00000010	3	00000000	3	00000001	3	00000001
4	00000010	4	00000000	4	00000001	4	00000001
Wr		Wr		Wr		Wr	
1	1	1	0	1	1	1	1
2	1	2	0	2	1	2	1
3	1	3	0	3	1	3	1
4	1	4	0	4	1	4	1
Rd		Rd		Rd		Rd	
1	0	1	1	1	1	1	1
2	0	2	1	2	1	2	1
3	0	3	1	3	1	3	1
4	0	4	1	4	1	4	1
Dataout		Dataout		Dataout		Dataout	
1	00000000	1	00000001	1	00000000	1	00000000
2	00000000	2	00000001	2	00000000	2	00000000
3	00000000	3	00000000	3	00000010	3	00000000

4	00000000	4	00000000	4	00000010	4	00000000
wclock	1	wclock	Don't care	wclock	1	wclock	1
rclock	Don't care	rclock	1	rclock	1	rclock	1
reset	0	reset	0	reset	0	reset	1

So, as shown in the table this is the expected outputs of router and the test strategy is as follows : when reading only , writing only , reading and writing or resetting. Unfortunately router was not giving the expected outputs.

7. Conclusion

NoC incorporates a wide range of examination, going from exceptionally theoretical programming related issues, across framework geography to actual level execution. This paper gives a review of cutting edge network-on-chip. The arising field of NoC examination and configuration challenges were talked about. Organization on Chip is a functioning exploration field with numerous viable applications in industry. This work centers around the framework, organization and connection level issues of the correspondence foundation. NoC research regions must be investigated to address the plan difficulties in cutting edge SOC framework that becomes pioneer in microelectronics.

The best implementation of FSM is three-process style that was implemented in the RR schedular.

From the test and simulation results and from the strategy used, resetting resets data outputs even if read and write requests are high, reading of a packet takes at least two clock cycles so that the schedular loops on it and be exited.

8. Synthesis Report

1. Register Module

- Device utilization summary:
  Selected Device : 7a100tcsg324-3
  Slice Logic Utilization:
  Slice Logic Distribution:

Number of LUT Flip Flop pairs used: 0
Number with an unused Flip Flop: 0 out of 0
Number with an unused LUT: 0 out of 0
Number of fully used LUT-FF pairs: 0 out of 0
Number of unique control sets: 1

- Timing Summary:
  Speed Grade: -3
  Minimum period: No path found
  Minimum input arrival time before clock: 0.661ns
  Maximum output required time after clock: 0.640ns
  Maximum combinational path delay: No path found

2. Demultiplexer Module

- Device utilization summary:
  Selected Device : 7a100tcsg324-3
  Slice Logic Utilization:
  Number of Slice LUTs: 4 out of 63400 0%
  Number used as Logic: 4 out of 63400 0%

  Slice Logic Distribution:
  Number of LUT Flip Flop pairs used: 4
  Number with an unused Flip Flop: 4 out of 4 100%
  Number with an unused LUT: 0 out of 4 0%
  Number of fully used LUT-FF pairs: 0 out of 4 0%
  Number of unique control sets: 4

- Timing Summary:
  Speed Grade: -3
  Minimum period: No path found
  Minimum input arrival time before clock: 0.898ns
  Maximum output required time after clock: 0.751ns
  Maximum combinational path delay: No path found

3. RAM Module

- Device utilization summary:

  Selected Device : 7a100tcsg324-3

  Slice Logic Utilization:

  Number of Slice LUTs: 8 out of 63400 0%

  Number used as Memory: 8 out of 19000 0%

  Number used as RAM: 8

  Slice Logic Distribution:

  Number of LUT Flip Flop pairs used: 8

  Number with an unused Flip Flop: 8 out of 8 100%

  Number with an unused LUT: 0 out of 8 0%

  Number of fully used LUT-FF pairs: 0 out of 8 0%

  Number of unique control sets: 1

- Timing Summary:

  Speed Grade: -3

  Minimum period: No path found

  Minimum input arrival time before clock: 0.885ns

  Maximum output required time after clock: 0.640ns

  Maximum combinational path delay: No path found

4. Gray Counter Module

- Device utilization summary:

  Selected Device : 7a100tcsg324-3

  Slice Logic Utilization:

  Number of Slice Registers: 3 out of 126800 0%

  Number of Slice LUTs: 4 out of 63400 0%

  Number used as Logic: 4 out of 63400 0%

  Slice Logic Distribution:

Number of LUT Flip Flop pairs used: 7
Number with an unused Flip Flop: 4 out of 7 57%
Number with an unused LUT: 3 out of 7 42%
Number of fully used LUT-FF pairs: 0 out of 7 0%
Number of unique control sets: 1

- Timing Summary:
  Speed Grade: -3
  Minimum period: 1.050ns (Maximum Frequency: 952.018MHz)
  Minimum input arrival time before clock: 0.639ns
  Maximum output required time after clock: 1.131ns
  Maximum combinational path delay: No path found

5. Gray to Binary Converter

- Device utilization summary:
  Selected Device : 7a100tcsg324-3
  Slice Logic Utilization:
  Number of Slice LUTs: 2 out of 63400 0%
  Number used as Logic: 2 out of 63400 0%

  Slice Logic Distribution:
  Number of LUT Flip Flop pairs used: 2
  Number with an unused Flip Flop: 2 out of 2 100%
  Number with an unused LUT: 0 out of 2 0%
  Number of fully used LUT-FF pairs: 0 out of 2 0%
  Number of unique control sets: 0

- Timing Summary:
  Speed Grade: -3
  Minimum period: No path found
  Minimum input arrival time before clock: No path found
  Maximum output required time after clock: No path found
  Maximum combinational path delay: 0.889ns

6. Controller Module

Device utilization summary:
Selected Device : 7a100tcsg324-3
Slice Logic Utilization:
Number of Slice Registers: 38 out of 126800 0%
Number of Slice LUTs: 57 out of 63400 0%
Number used as Logic: 57 out of 63400 0%

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 71
Number with an unused Flip Flop: 33 out of 71 46%
Number with an unused LUT: 14 out of 71 19%
Number of fully used LUT-FF pairs: 24 out of 71 33%
Number of unique control sets: 34

7. FIFO Module

Device utilization summary:
Selected Device : 7a100tcsg324-3
Slice Logic Utilization:
Number of Slice Registers: 38 out of 126800 0%
Number of Slice LUTs: 53 out of 63400 0%
Number used as Logic: 45 out of 63400 0%
Number used as Memory: 8 out of 19000 0%
Number used as RAM: 8

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 68
Number with an unused Flip Flop: 30 out of 68 44%
Number with an unused LUT: 15 out of 68 22%
Number of fully used LUT-FF pairs: 23 out of 68 33%
Number of unique control sets: 29

Timing Summary:
Speed Grade: -3
Minimum period: 2.197ns (Maximum Frequency: 455.249MHz)
Minimum input arrival time before clock: 1.530ns
Maximum output required time after clock: 2.056ns
Maximum combinational path delay: 1.177ns

8. Round Robin Module

Device utilization summary:

Selected Device : 7a100tcsg324-3
Slice Logic Utilization:
Number of Slice LUTs: 8 out of 63400 0%
Number used as Logic: 8 out of 63400 0%
Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 8
Number with an unused Flip Flop: 8 out of 8 100%
Number with an unused LUT: 0 out of 8 0%
Number of fully used LUT-FF pairs: 0 out of 8 0%
Number of unique control sets: 0

Timing Summary(analysis):
Speed Grade: -3
Minimum period: No path found
Minimum input arrival time before clock: No path found
Maximum output required time after clock: No path found
Maximum combinational path delay: 0.921ns

9. 4-Port Router Module

Device utilization summary:

Selected Device : 7a100tcsg324-3
Slice Logic Utilization:
Number of Slice Registers: 802 out of 126800 0%
Number of Slice LUTs: 866 out of 63400 1%
Number used as Logic: 738 out of 63400 1%
Number used as Memory: 128 out of 19000 0%
Number used as RAM: 128

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 1248
Number with an unused Flip Flop: 446 out of 1248 35%
Number with an unused LUT: 382 out of 1248 30%
Number of fully used LUT-FF pairs: 420 out of 1248 33%
Number of unique control sets: 459

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 574
Number with an unused Flip Flop: 156 out of 574 27%
Number with an unused LUT: 228 out of 574 39%
Number of fully used LUT-FF pairs: 190 out of 574 33%
Number of unique control sets: 73

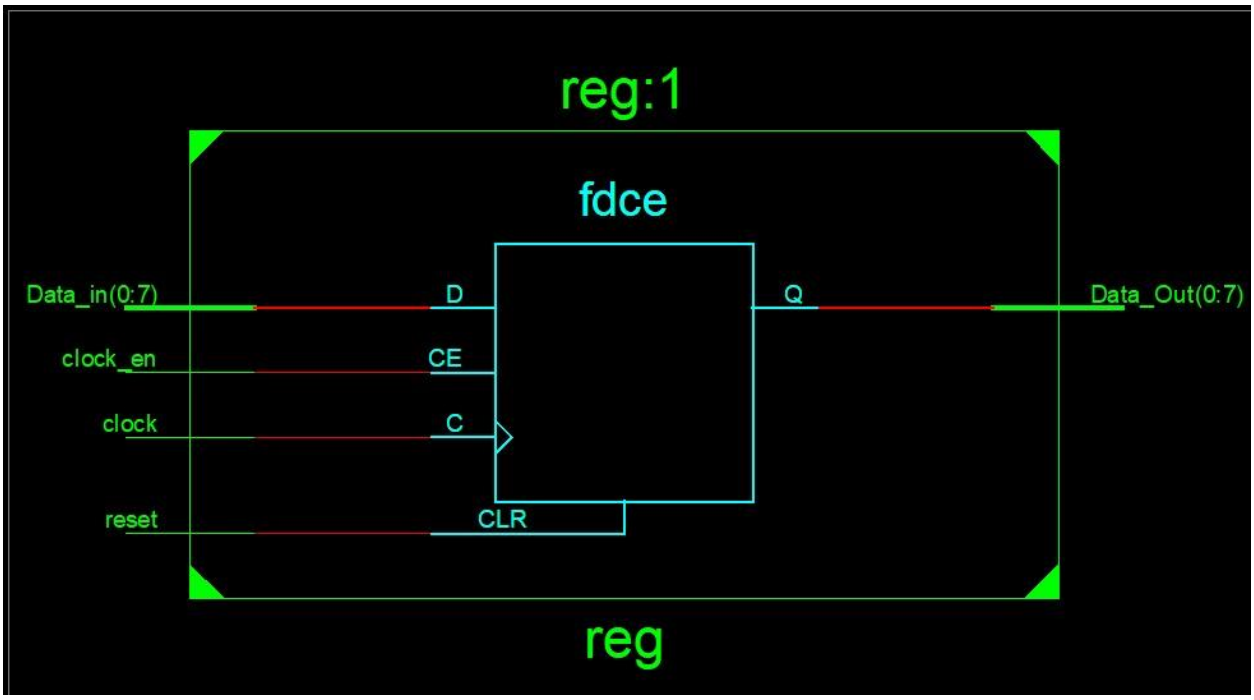
Timing Summary:
Speed Grade: -3
Minimum period: 2.179ns (Maximum Frequency: 458.905MHz)
Minimum input arrival time before clock: 1.757ns
Maximum output required time after clock: 0.640ns
Maximum combinational path delay: No path found

Appendix A

- **Module 1**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY reg IS
PORT( clock, clock_en,reset: IN std_logic;
    Data_in: IN std_logic_vector (7 downto 0);
    Data_Out: OUT std_logic_vector (7 downto 0));
END ENTITY reg;
ARCHITECTURE behav OF reg IS
BEGIN

process(clock,clock_en,reset) IS
BEGIN
IF Reset = '1' THEN
Data_Out <= "00000000" ;
ELSIF rising_edge(clock) AND clock_en = '1' THEN
Data_out <= Data_in ;
else
Data_out<="00000000" ;
```

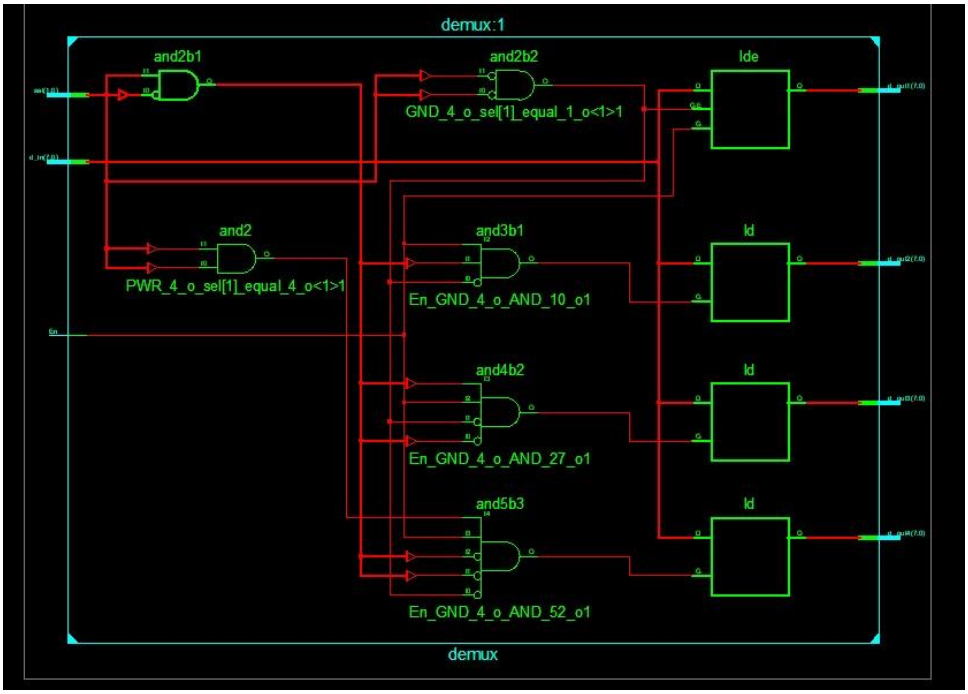
```
end if;
END PROCESS;

END ARCHITECTURE behav;
```



- **Module 2**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY demux IS
PORT( sel: IN std_logic_vector (1 DOWNTO 0);
En: IN std_logic;
d_in: IN std_logic_vector(7 DOWNTO 0);
d_out1: OUT std_logic_vector(7 DOWNTO 0);
d_out2: OUT std_logic_vector(7 DOWNTO 0);
d_out3: OUT std_logic_vector(7 DOWNTO 0);
d_out4: OUT std_logic_vector (7 DOWNTO 0));
```

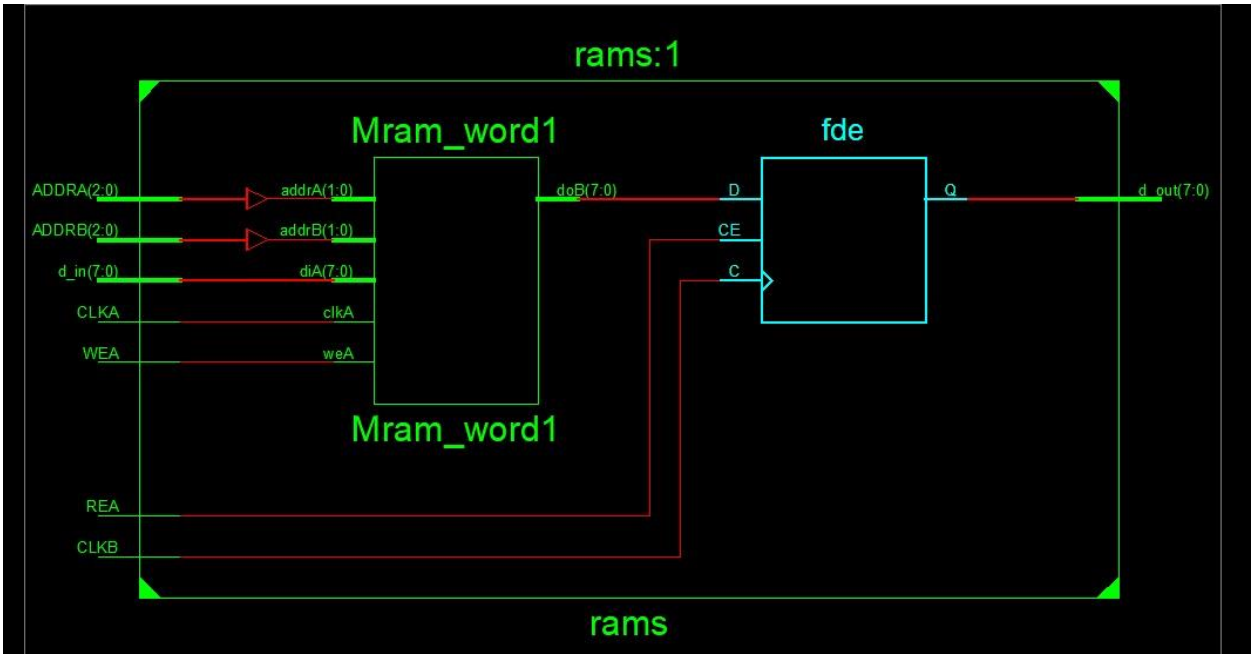
```
END ENTITY demux;
ARCHITECTURE omas OF demux IS
BEGIN
dm: PROCESS (sel, d_in) IS

BEGIN


IF En= '1' THEN
IF sel="00" THEN
d_out1<=d_in;
ELSIF sel="01" THEN
d_out2<=d_in;
ELSIF sel="10" THEN
d_out3<=d_in;
ELSIF sel="11" THEN
d_out4<=d_in;

END IF;
END IF;


END PROCESS dm;
END ARCHITECTURE omas;
```



- **Module 3**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY rams IS
        port(d_in: in std_logic_vector(7 downto 0);
            d_out: out std_logic_vector(7 downto 0);
            WEA: in std_logic;
            REA: in std_logic;
            ADDRA: in std_logic_vector(2 downto 0);
            ADDRB: in std_logic_vector(2 downto 0);
            CLKA: in std_logic;
            CLKB: in std_logic);
END ENTITY rams;

ARCHITECTURE ARCH_M_ROU_03 of rams is
```

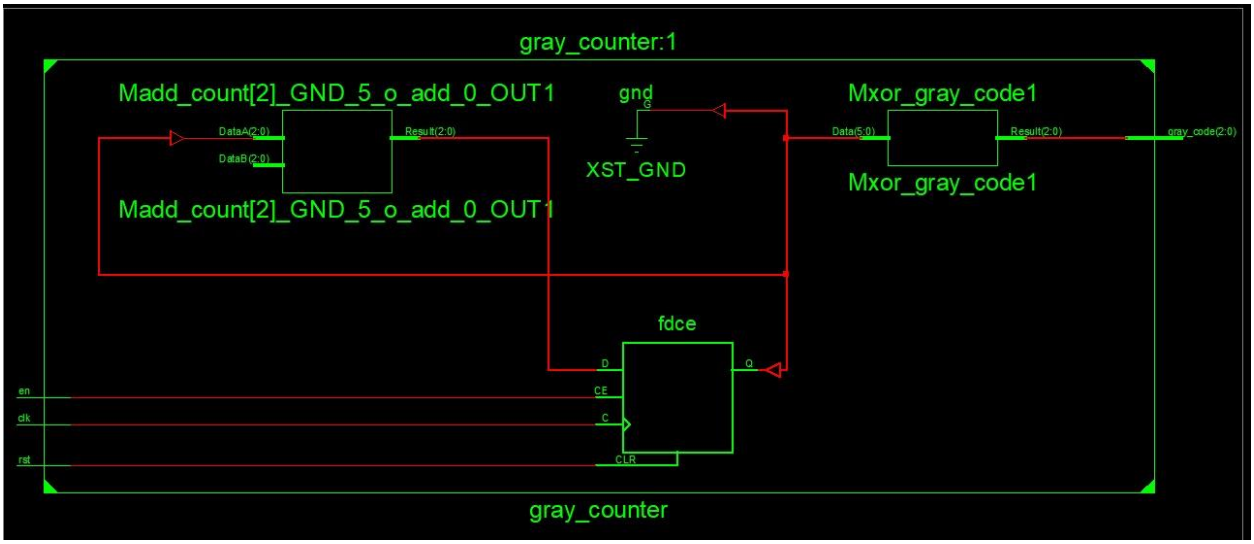
```
        type rm is array (2 downto 0) of std_logic_vector (7 downto 0);
        signal word: rm;


begin
        P1: process(CLKA)
        begin
                if( CLKA' event and CLKA='1' and WEA='1') then
                                word(conv_integer(ADDRA))<= d_in; --write
                end if;
        end process  P1;

        P2: process(CLKB)
        begin
                if(CLKB' event and CLKB='1' and REA='1') then
                                d_out <= word(conv_integer(ADDRB)); -- read
                elsif(REA ='0') then
                        d_out<="00000000";
                end if;

        end process  P2;

END ARCHITECTURE ARCH_M_ROU_03;
```



- **Module 4**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity gray_counter is
port(clk: in std_logic;
    rst: in std_logic;
    en:  in std_logic;
    gray_code: out std_logic_vector (2 downto 0));

end gray_counter;

architecture behav of gray_counter is
signal count : std_logic_vector(2 downto 0) := "000";

begin
```

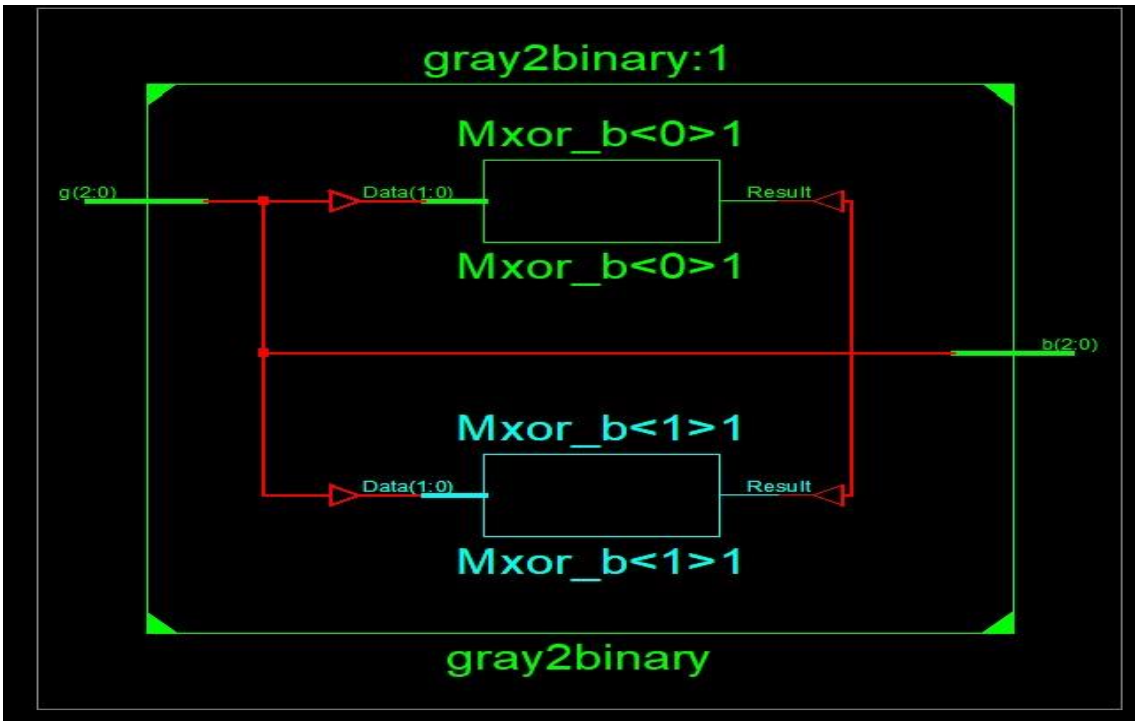
```
process(clk)
begin

 if (rst='1') then
  count <= "000";
 elsif (rising_edge(clk) AND en = '1') then
  count <= count + "001";
 end if;
end process;
   gray_code <= count xor ('0' & count(2 downto 1));

end behav;
```



- **Module 5**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity gray2binary is
Port ( g : in STD_LOGIC_VECTOR (2 downto 0);
```

```
      b : out STD_LOGIC_VECTOR (2 downto 0));
end gray2binary;

architecture Behavioral1 of gray2binary is
begin
b(2)<= g(2);
b(1)<= g(2) xor g(1);
b(0)<= g(2) xor g(1) xor g(0);
end Behavioral1;
```



- **Module 6**

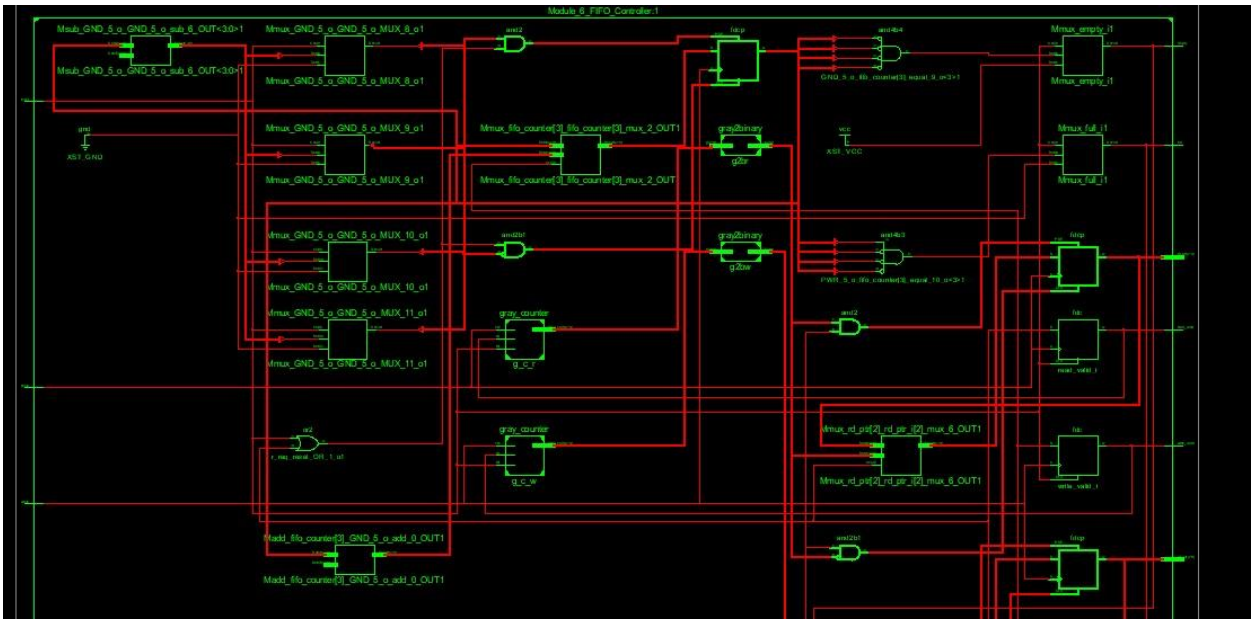
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

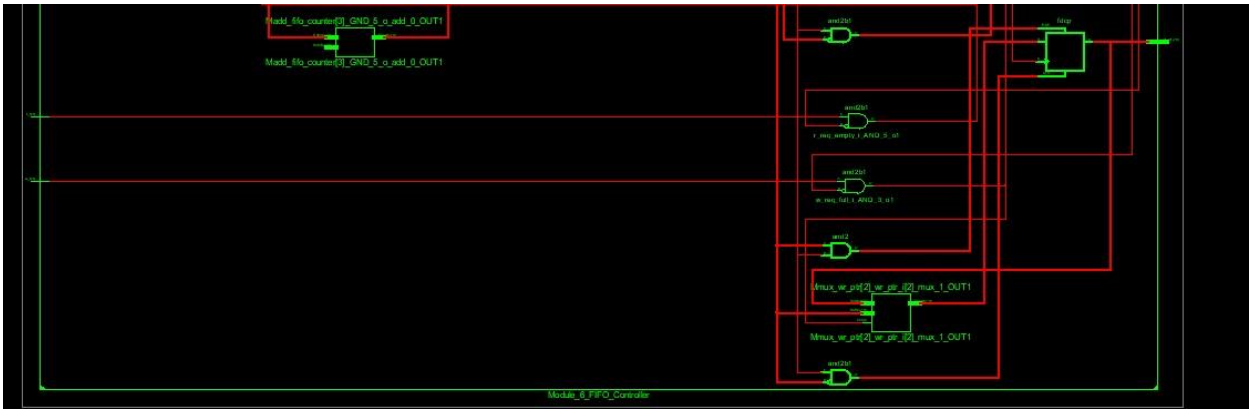
```
entity Module_6_FIFO_Controller is
   Port ( reset : in  STD_LOGIC;
          rdclk : in  STD_LOGIC;
          wrclk : in  STD_LOGIC;
          r_req : in  STD_LOGIC;
          w_req : in  STD_LOGIC;
          write_valid : out  STD_LOGIC;
          read_valid : out  STD_LOGIC;
                          wr_ptr : out  STD_LOGIC_VECTOR (2 downto 0);
          rd_ptr : out  STD_LOGIC_VECTOR (2 downto 0);
          empty : out  STD_LOGIC;
          full : out  STD_LOGIC);
end Module_6_FIFO_Controller;

architecture Behavioral of Module_6_FIFO_Controller is


component gray_counter is
port(clk: in std_logic;
    rst: in std_logic;
    en:  in std_logic;
    gray_code: out std_logic_vector (2 downto 0));

end component gray_counter;

For g_c_w : gray_counter use entity WORK.gray_counter (behav);
For g_c_r : gray_counter use entity WORK.gray_counter (behav);


component gray2binary is
Port ( g : in STD_LOGIC_VECTOR (2 downto 0);
      b : out STD_LOGIC_VECTOR (2 downto 0));
end component gray2binary;

For g2bw : gray2binary use entity WORK.gray2binary (Behavioral1);
```

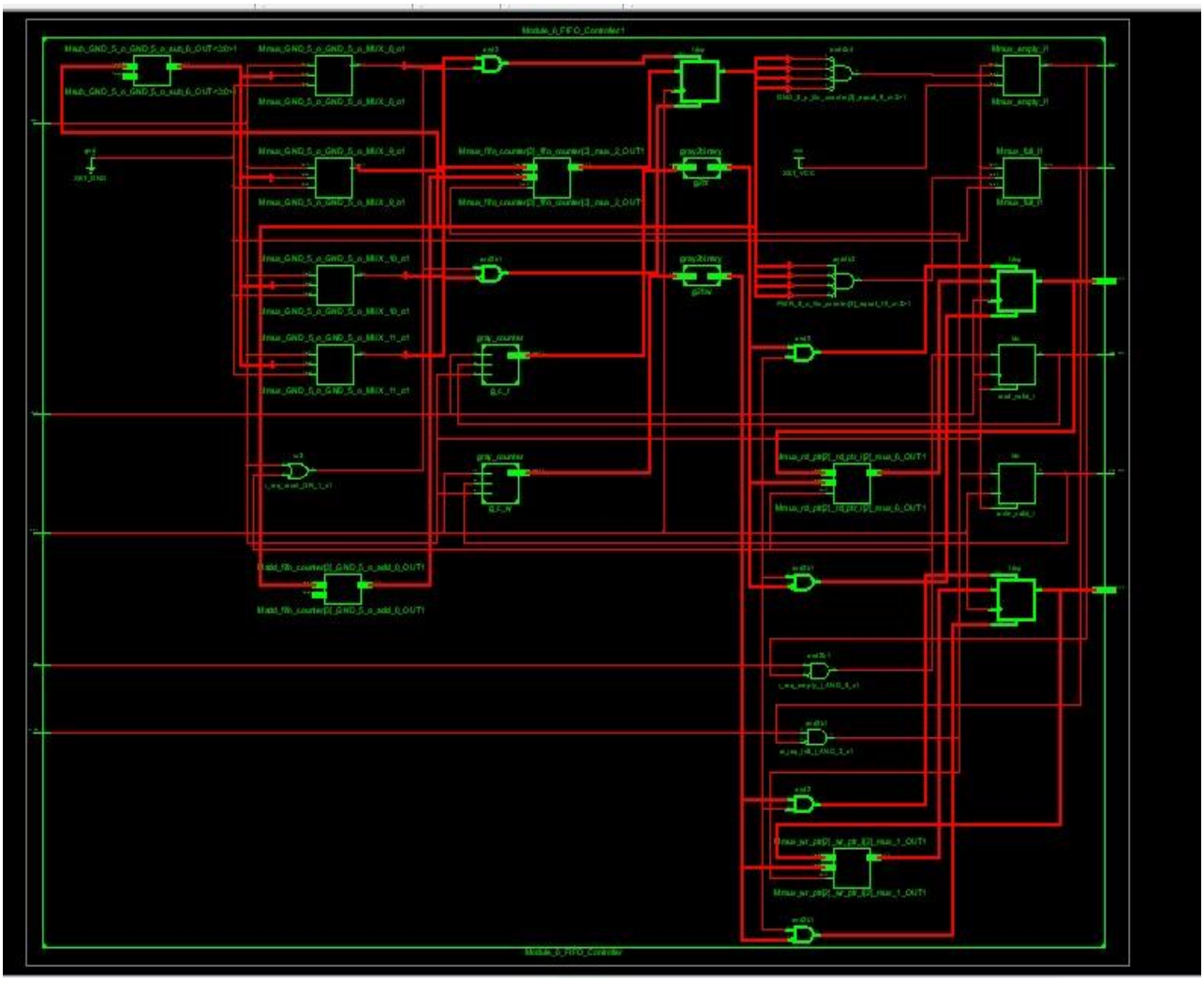
```
For g2br : gray2binary use entity WORK.gray2binary (Behavioral1);

signal grayw : std_logic_vector (2 downto 0);
signal grayr : std_logic_vector (2 downto 0);
signal write_valid_i : STD_LOGIC;
signal read_valid_i : STD_LOGIC;
signal wr_ptr_i : STD_LOGIC_VECTOR (2 downto 0);
signal rd_ptr_i : STD_LOGIC_VECTOR (2 downto 0);
signal empty_i : STD_LOGIC:='1';
signal full_i : STD_LOGIC:='0';
signal fifo_counter: integer range 0 to 8 := 0 ;
constant fifo_max: integer := 8 ;


begin

--Assigning signals to outputs
write_valid <= write_valid_i;
read_valid <= read_valid_i;


empty <= empty_i;
full <= full_i;


--updating empty and full

--preparing wr_ptr
g_c_w : gray_counter port map(wrclk , reset , write_valid_i , grayw);
g2bw : gray2binary port map (grayw,wr_ptr_i);


--preparing rd_ptr
g_c_r : gray_counter port map (rdclk , reset , read_valid_i , grayr );
g2br : gray2binary port map (grayr,rd_ptr_i) ;
```

```
--Checking Validity and assigning wrptr and rdptr and updating fifo_counter
validity : process (reset,wrclk,rdclk,r_req,w_req,empty_i,full_i)
begin
if reset='1' then
                write_valid_i <= '0';
                read_valid_i<='0';
                wr_ptr <= wr_ptr_i;
                rd_ptr <= rd_ptr_i;
                fifo_counter<= 0;

else
        if rising_edge(wrclk) then
                if w_req='1' and full_i='0' then
                                wr_ptr <= wr_ptr_i;
                                fifo_counter<= fifo_counter + 1;
                                write_valid_i<='1';
                else
                                write_valid_i<='0';
                end if;
        end if;

        if rising_edge(rdclk) then
                if r_req='1' and empty_i='0' then
                                rd_ptr <= rd_ptr_i;
                                fifo_counter<= fifo_counter - 1;
                                read_valid_i <= '1';
                else
                                read_valid_i<='0';
                end if;
        end if;
end if;
end process;
```

```
--updating empty and full
p:process(reset,fifo_counter)
begin
if (reset='1') then
                empty_i<='1';
                full_i<='0';
else

        if (fifo_counter=0) then
                        empty_i <= '1';
        else
                        empty_i <= '0';
        end if;

        if (fifo_counter=fifo_max) then
                        full_i <='1';
        else
                        full_i <='0';
        end if;
end if;
end process;


end Behavioral;
```


Module_6_FIFO_Controller:1
Msub_GND_5_o_GND_5_o_sub_6_OUT<3:0>1
Mmux_GND_5_o_GND_5_o_MUX_8_o1
Mmux_GND_5_o_GND_5_o_MUX_9_o1
Mmux_GND_5_o_GND_5_o_MUX_10_o1
Mmux_GND_5_o_GND_5_o_MUX_11_o1
Mmux_fifo_counter[3]_fifo_counter[3]_mux_2_OUT1
Mmux_fifo_counter[3]_fifo_counter[3]_mux_2_OUT
gray2binary
g2br
gray_counter
g_c_r
g_c_w
Mmux_rd_ptr[2]_rd_ptr_i[2]_mux_6_OUT1
Mmux_empty_i1
Mmux_full_i1
Madd_fifo_counter[3]_GND_5_o_add_0_OUT1


Madd_fifo_counter[3]_GND_5_o_add_0_OUT1
Mmux_wr_ptr[2]_wr_ptr_i[2]_mux_1_OUT1
Module_6_FIFO_Controller

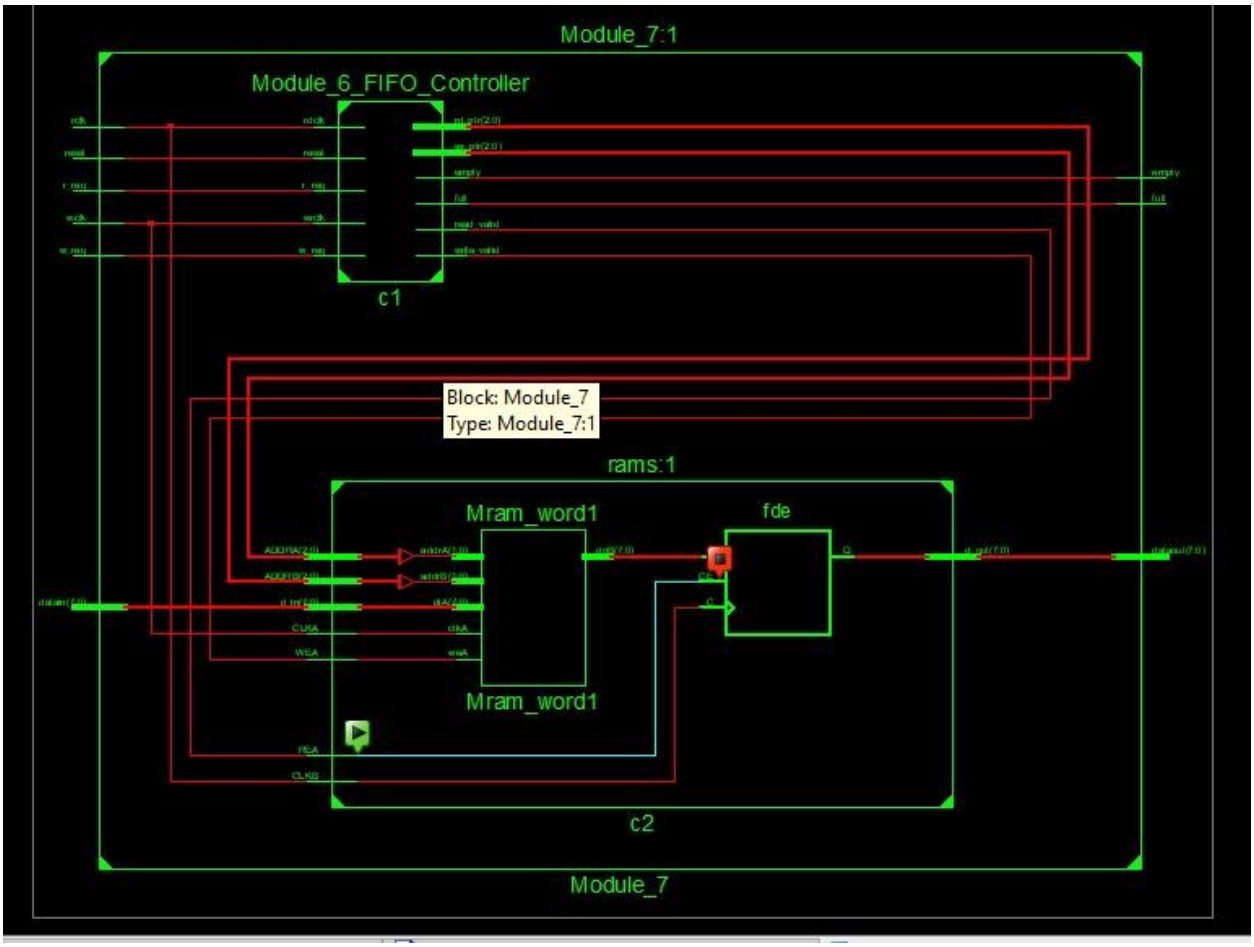


- **Module 7**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Module_7 is
    Port ( reset : in  STD_LOGIC;
           rclk : in  STD_LOGIC;
           wclk : in  STD_LOGIC;
           r_req : in  STD_LOGIC;
```

```
        w_req : in  STD_LOGIC;
        datain : in  STD_LOGIC_VECTOR (7 downto 0);
        dataout : out  STD_LOGIC_VECTOR (7 downto 0);
        empty : out  STD_LOGIC;
        full : out  STD_LOGIC);
end Module_7;

architecture Behavioral of Module_7  is

component Module_6_FIFO_Controller is
   Port ( reset : in  STD_LOGIC;
        rdclk : in  STD_LOGIC;
        wrclk : in  STD_LOGIC;
        r_req : in  STD_LOGIC;
        w_req : in  STD_LOGIC;
        write_valid : out  STD_LOGIC;
        read_valid : out  STD_LOGIC;
                        wr_ptr : out  STD_LOGIC_VECTOR (2 downto 0);
        rd_ptr : out  STD_LOGIC_VECTOR (2 downto 0);
        empty : out  STD_LOGIC;
        full : out  STD_LOGIC);
end component Module_6_FIFO_Controller;

For c1: Module_6_FIFO_Controller use entity
work.Module_6_FIFO_Controller (Behavioral);

component rams IS
        port(d_in: in std_logic_vector(7 downto 0);
            d_out: out std_logic_vector(7 downto 0);
            WEA: in std_logic;
            REA: in std_logic;
            ADDRA: in std_logic_vector(2 downto 0);
            ADDRB: in std_logic_vector(2 downto 0);
            CLKA: in std_logic;
            CLKB: in std_logic);
```

```
END component rams;

for c2: rams use entity work.rams (ARCH_M_ROU_03);

signal wrvi: STD_LOGIC;
signal rdvi: STD_LOGIC;
signal wptri: STD_LOGIC_VECTOR (2 downto 0);
signal rptri: STD_LOGIC_VECTOR (2 downto 0);
signal empty_i: STD_LOGIC;
signal full_i: STD_LOGIC;
signal dataout_i:STD_LOGIC_VECTOR (7 downto 0);
signal wclk_i: std_logic;
signal rclk_i: std_logic;

begin


empty<= empty_i;
full<= full_i;
dataout<=dataout_i;

c1: Module_6_FIFO_Controller port map
(reset,rclk,wclk,r_req,w_req,wrvi,rdvi,wptri,rptri,empty_i,full_i) ;

c2: rams port
map(datain,dataout_i,wrvi,rdvi,wptri,rptri,CLKA=>wrclk,CLKB=>rdclk);
end Behavioral;
```

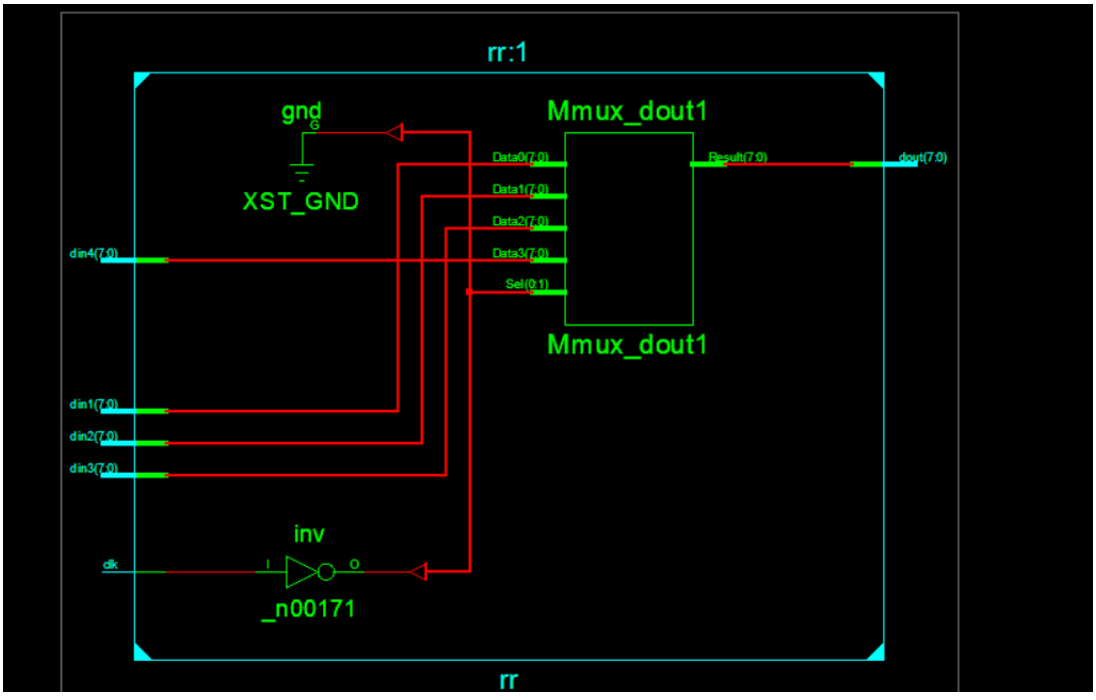


- **Module 8**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
Use IEEE.STD_LOGIC_ARITH.ALL;
Use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- for the to integer function
ENTITY rr IS
PORT( din1: IN std_logic_vector (7 DOWNTO 0);
din2: IN std_logic_vector (7 DOWNTO 0);
din3: IN std_logic_vector (7 DOWNTO 0);
din4: IN std_logic_vector (7 DOWNTO 0);
```

```
clk: IN std_logic;
dout: OUT std_logic_vector (7 DOWNTO 0));
END ENTITY rr;
ARCHITECTURE behav OF rr IS
TYPE state IS (a,b,c,d);
SIGNAL cs : state;
SIGNAL ns : state;
BEGIN
S: PROCESS (clk) IS
BEGIN
IF clk'event and clk = '1' THEN
cs<=a;
ELSE
cs<=b;
end if;
END PROCESS S;

SS: PROCESS (cs) IS
BEGIN
CASE cs IS

WHEN a =>
ns <= b;

WHEN b =>
ns <= c;
WHEN c =>
ns <= d;

WHEN d =>
ns <= a;

END CASE;

END PROCESS SS;
```

```
SSS: PROCESS (cs,din1,din2,din3,din4) IS
BEGIN
IF cs = a THEN
dout <= din1;

ELSIF cs = b THEN
dout <= din2;

ELSIF cs = c THEN
dout <= din3;

ELSE
dout <= din4;

end if;

END PROCESS SSS;


END ARCHITECTURE behav;
```



- **Module 9**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity Module_9 is
    Port ( datai1 : in  STD_LOGIC_VECTOR (7 downto 0);
           datai2 : in  STD_LOGIC_VECTOR (7 downto 0);
           datai3 : in  STD_LOGIC_VECTOR (7 downto 0);
           datai4 : in  STD_LOGIC_VECTOR (7 downto 0);
           wr1 : in  STD_LOGIC;
           wr2 : in  STD_LOGIC;
           wr3 : in  STD_LOGIC;
           wr4 : in  STD_LOGIC;
                      rd1 : in  STD_LOGIC;
           rd2 : in  STD_LOGIC;
           rd3 : in  STD_LOGIC;
           rd4 : in  STD_LOGIC;
           wclock : in  STD_LOGIC;
           rclock : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           datao1 : out  STD_LOGIC_VECTOR (7 downto 0);
           datao2 : out  STD_LOGIC_VECTOR (7 downto 0);
           datao3 : out  STD_LOGIC_VECTOR (7 downto 0);
           datao4 : out  STD_LOGIC_VECTOR (7 downto 0));
end Module_9;


architecture Behave of Module_9 is
```

```
component demux IS
PORT( sel: IN std_logic_vector (1 DOWNTO 0);
En: IN std_logic;
d_in: IN std_logic_vector(7 DOWNTO 0);
d_out1: OUT std_logic_vector(7 DOWNTO 0);
d_out2: OUT std_logic_vector(7 DOWNTO 0);
d_out3: OUT std_logic_vector(7 DOWNTO 0);
d_out4: OUT std_logic_vector (7 DOWNTO 0));
END component demux;

For demux1: demux use entity work.demux (omas);
For demux2: demux use entity work.demux (omas);
For demux3: demux use entity work.demux (omas);
For demux4: demux use entity work.demux (omas);

--signals of demux
--demux1
signal d_out1_1:  std_logic_vector(7 DOWNTO 0);
signal d_out2_1:  std_logic_vector(7 DOWNTO 0);
signal d_out3_1:  std_logic_vector(7 DOWNTO 0);
signal d_out4_1:  std_logic_vector(7 DOWNTO 0);
--demux2
signal d_out1_2:  std_logic_vector(7 DOWNTO 0);
signal d_out2_2:  std_logic_vector(7 DOWNTO 0);
signal d_out3_2:  std_logic_vector(7 DOWNTO 0);
signal d_out4_2:  std_logic_vector(7 DOWNTO 0);
--demux3
signal d_out1_3:  std_logic_vector(7 DOWNTO 0);
signal d_out2_3:  std_logic_vector(7 DOWNTO 0);
```

```
signal d_out3_3:  std_logic_vector(7 DOWNTO 0);
signal d_out4_3:  std_logic_vector(7 DOWNTO 0);
--demux4
signal d_out1_4:  std_logic_vector(7 DOWNTO 0);
signal d_out2_4:  std_logic_vector(7 DOWNTO 0);
signal d_out3_4:  std_logic_vector(7 DOWNTO 0);
signal d_out4_4:  std_logic_vector(7 DOWNTO 0);
-----------------


component reg IS
PORT( clock, clock_en,reset: IN std_logic;
     Data_in: IN std_logic_vector (7 downto 0);
     Data_Out: OUT std_logic_vector (7 downto 0));
END component reg;

For reg1: reg use entity work.reg (behav);
For reg2: reg use entity work.reg (behav);
For reg3: reg use entity work.reg (behav);
For reg4: reg use entity work.reg (behav);
For reg5: reg use entity work.reg (behav);
For reg6: reg use entity work.reg (behav);
For reg7: reg use entity work.reg (behav);
For reg8: reg use entity work.reg (behav);

--signals of registers
signal Data_Out1: std_logic_vector (7 downto 0);
signal Data_Out2: std_logic_vector (7 downto 0);
signal Data_Out3: std_logic_vector (7 downto 0);
signal Data_Out4: std_logic_vector (7 downto 0);
```

```
signal Data_Out5: std_logic_vector (7 downto 0);
signal Data_Out6: std_logic_vector (7 downto 0);
signal Data_Out7: std_logic_vector (7 downto 0);
signal Data_Out8: std_logic_vector (7 downto 0);
----------------------

component rr IS
PORT( din1: IN std_logic_vector (7 DOWNTO 0);
din2: IN std_logic_vector (7 DOWNTO 0);
din3: IN std_logic_vector (7 DOWNTO 0);
din4: IN std_logic_vector (7 DOWNTO 0);
clk: IN std_logic;
dout: OUT std_logic_vector (7 DOWNTO 0));
END component rr;

For rr1: rr use entity work.rr (behav);
For rr2: rr use entity work.rr (behav);
For rr3: rr use entity work.rr (behav);
For rr4: rr use entity work.rr (behav);

--signals of rr
signal dout1:  std_logic_vector (7 DOWNTO 0);
signal dout2:  std_logic_vector (7 DOWNTO 0);
signal dout3:  std_logic_vector (7 DOWNTO 0);
signal dout4:  std_logic_vector (7 DOWNTO 0);
---------------

component Module_7 is
   Port ( reset : in  STD_LOGIC;
          rclk : in  STD_LOGIC;
```

```
        wclk : in  STD_LOGIC;
        r_req : in  STD_LOGIC;
        w_req : in  STD_LOGIC;
        datain : in  STD_LOGIC_VECTOR (7 downto 0);
        dataout : out  STD_LOGIC_VECTOR (7 downto 0);
        empty : out  STD_LOGIC;
        full : out  STD_LOGIC);
end component Module_7;

For fifo1: Module_7 use entity work.Module_7 (Behavioral);
For fifo2: Module_7 use entity work.Module_7 (Behavioral);
For fifo3: Module_7 use entity work.Module_7 (Behavioral);
For fifo4: Module_7 use entity work.Module_7 (Behavioral);
For fifo5: Module_7 use entity work.Module_7 (Behavioral);
For fifo6: Module_7 use entity work.Module_7 (Behavioral);
For fifo7: Module_7 use entity work.Module_7 (Behavioral);
For fifo8: Module_7 use entity work.Module_7 (Behavioral);
For fifo9: Module_7 use entity work.Module_7 (Behavioral);
For fifo10: Module_7 use entity work.Module_7 (Behavioral);
For fifo11: Module_7 use entity work.Module_7 (Behavioral);
For fifo12: Module_7 use entity work.Module_7 (Behavioral);
For fifo13: Module_7 use entity work.Module_7 (Behavioral);
For fifo14: Module_7 use entity work.Module_7 (Behavioral);
For fifo15: Module_7 use entity work.Module_7 (Behavioral);
For fifo_16: Module_7 use entity work.Module_7 (Behavioral);

--signals of fifo

--fifo1
signal dataout1 : STD_LOGIC_VECTOR (7 downto 0);
```

```
signal empty1 :   STD_LOGIC;
signal full1 :    STD_LOGIC;
--fifo2
signal dataout2 : STD_LOGIC_VECTOR (7 downto 0);
signal empty2 :   STD_LOGIC;
signal full2 :    STD_LOGIC;
--fifo3
signal dataout3 : STD_LOGIC_VECTOR (7 downto 0);
signal empty3 :   STD_LOGIC;
signal full3 :    STD_LOGIC;
--fifo4
signal dataout4 : STD_LOGIC_VECTOR (7 downto 0);
signal empty4 :   STD_LOGIC;
signal full4 :    STD_LOGIC;
--fifo5
signal dataout5 : STD_LOGIC_VECTOR (7 downto 0);
signal empty5 :   STD_LOGIC;
signal full5 :    STD_LOGIC;
--fifo6
signal dataout6 : STD_LOGIC_VECTOR (7 downto 0);
signal empty6 :   STD_LOGIC;
signal full6 :    STD_LOGIC;
--fifo7
signal dataout7 : STD_LOGIC_VECTOR (7 downto 0);
signal empty7 :   STD_LOGIC;
signal full7 :    STD_LOGIC;
--fifo8
signal dataout8 : STD_LOGIC_VECTOR (7 downto 0);
signal empty8 :   STD_LOGIC;
signal full8 :    STD_LOGIC;
```

```
--fifo9
signal dataout9 : STD_LOGIC_VECTOR (7 downto 0);
signal empty9 :   STD_LOGIC;
signal full9 :    STD_LOGIC;
--fifo10
signal dataout10 : STD_LOGIC_VECTOR (7 downto 0);
signal empty10 :   STD_LOGIC;
signal full10 :    STD_LOGIC;
--fifo11
signal dataout11 : STD_LOGIC_VECTOR (7 downto 0);
signal empty11 :   STD_LOGIC;
signal full11 :    STD_LOGIC;
--fifo12
signal dataout12 : STD_LOGIC_VECTOR (7 downto 0);
signal empty12 :   STD_LOGIC;
signal full12 :    STD_LOGIC;
--fifo13
signal dataout13 : STD_LOGIC_VECTOR (7 downto 0);
signal empty13 :   STD_LOGIC;
signal full13 :    STD_LOGIC;
--fifo14
signal dataout14 : STD_LOGIC_VECTOR (7 downto 0);
signal empty14 :   STD_LOGIC;
signal full14 :    STD_LOGIC;
--fifo15
signal dataout15 : STD_LOGIC_VECTOR (7 downto 0);
signal empty15 :   STD_LOGIC;
signal full15 :    STD_LOGIC;
--fifo16
signal dataout16 : STD_LOGIC_VECTOR (7 downto 0);
```

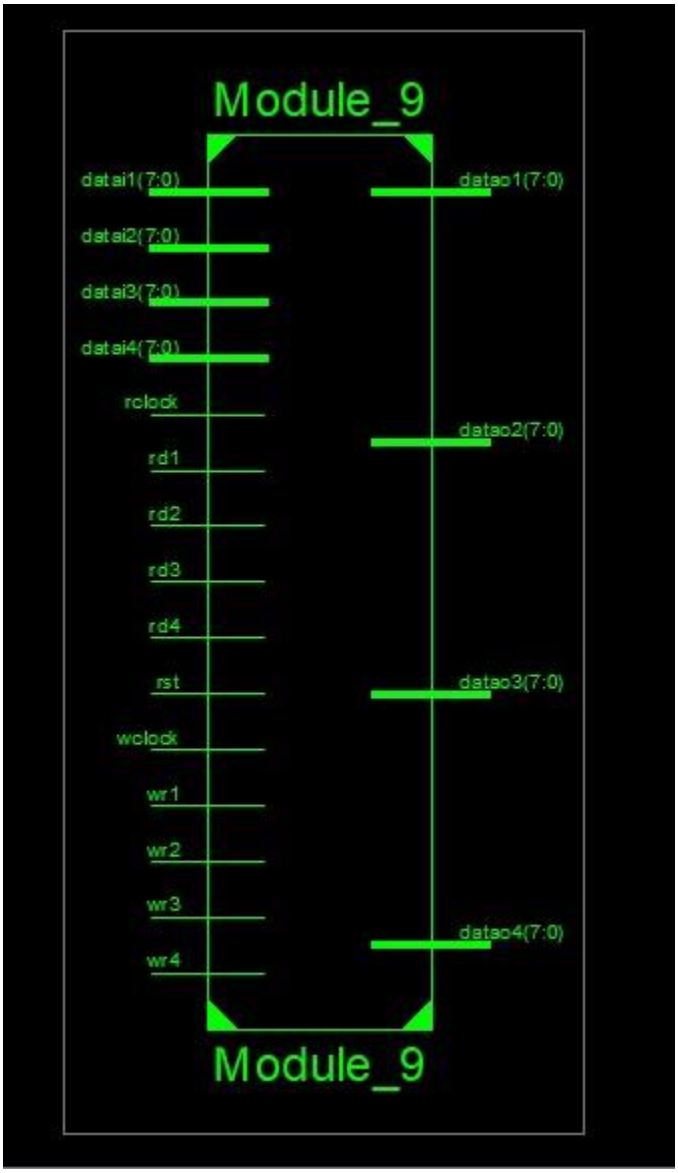
```
signal empty16 :   STD_LOGIC;
signal full16 :    STD_LOGIC;



begin
--Assigning dataout siganls to dataout
datao1<=Data_Out5;
datao2<=Data_Out6;
datao3<=Data_Out7;
datao4<=Data_Out8;


--Port mapping of input buffers registers
reg1: reg port map(wclock,wr1,rst,datai1,Data_Out1);
reg2: reg port map(wclock,wr2,rst,datai2,Data_Out2);
reg3: reg port map(wclock,wr3,rst,datai3,Data_Out3);
reg4: reg port map(wclock,wr4,rst,datai4,Data_Out4);


--Port mapping of demultiplexers (with input buffers)
demux1: demux port map(Data_Out1(1 downto
0),wr1,Data_Out1,d_out1_1,d_out2_1,d_out3_1,d_out4_1);
demux2: demux port map(Data_Out2(1 downto
0),wr2,Data_Out2,d_out1_2,d_out2_2,d_out3_2,d_out4_2);
demux3: demux port map(Data_Out3(1 downto
0),wr3,Data_Out3,d_out1_3,d_out2_3,d_out3_3,d_out4_3);
demux4: demux port map(Data_Out4(1 downto
0),wr4,Data_Out4,d_out1_4,d_out2_4,d_out3_4,d_out4_4);
```

```
--Port mapping of FIFO components (inputs are outputs of
Demuxs)
--Takes dataout 1 of each Demux
--Takes wr 1,2,3,4 respectively
fifo1: Module_7 port
map(rst,rclock,wclock,rd1,wr1,d_out1_1,dataout1,empty1,full1);
fifo2: Module_7 port
map(rst,rclock,wclock,rd2,wr2,d_out1_2,dataout2,empty2,full2);
fifo3: Module_7 port
map(rst,rclock,wclock,rd3,wr3,d_out1_3,dataout3,empty3,full3);
fifo4: Module_7 port
map(rst,rclock,wclock,rd4,wr4,d_out1_4,dataout4,empty4,full4);

fifo5: Module_7 port
map(rst,rclock,wclock,rd1,wr1,d_out2_1,dataout5,empty5,full5);
fifo6: Module_7 port
map(rst,rclock,wclock,rd2,wr2,d_out2_2,dataout6,empty6,full6);
fifo7: Module_7 port
map(rst,rclock,wclock,rd3,wr3,d_out2_3,dataout7,empty7,full7);
fifo8: Module_7 port
map(rst,rclock,wclock,rd4,wr4,d_out2_4,dataout8,empty8,full8);

fifo9: Module_7 port
map(rst,rclock,wclock,rd1,wr1,d_out3_1,dataout9,empty9,full9);
fifo10: Module_7 port
map(rst,rclock,wclock,rd2,wr2,d_out3_2,dataout10,empty10,full1
0);
fifo11: Module_7 port
map(rst,rclock,wclock,rd3,wr3,d_out3_3,dataout11,empty11,full1
1);
```

```
fifo12: Module_7 port
map(rst,rclock,wclock,rd4,wr4,d_out3_4,dataout12,empty12,full12);

fifo13: Module_7 port
map(rst,rclock,wclock,rd1,wr1,d_out4_1,dataout13,empty13,full13);
fifo14: Module_7 port
map(rst,rclock,wclock,rd2,wr2,d_out4_2,dataout14,empty14,full14);
fifo15: Module_7 port
map(rst,rclock,wclock,rd3,wr3,d_out4_3,dataout15,empty15,full15);
fifo_16: Module_7 port
map(rst,rclock,wclock,rd4,wr4,d_out4_4,dataout16,empty16,full16);


--Port mapping of RR components (inputs of RR1 are outputs of fifo 1,5,9,13 & wr1)
rr1: rr port
map(dataout1,dataout5,dataout9,dataout13,rclock,dout1);
rr2: rr port
map(dataout2,dataout6,dataout10,dataout14,rclock,dout2);
rr3: rr port
map(dataout3,dataout7,dataout11,dataout15,rclock,dout3);
rr4: rr port
map(dataout4,dataout8,dataout12,dataout16,rclock,dout4);

--Port mapping of output buffers registers
```

reg5: reg port map(rclock,rd1,rst,dout1,Data_Out5);
reg6: reg port map(rclock,rd2,rst,dout2,Data_Out6);
reg7: reg port map(rclock,rd3,rst,dout3,Data_Out7);
reg8: reg port map(rclock,rd4,rst,dout4,Data_Out8);

end Behave;



Appendix B

- **Module 10 (Testbench of module 9)**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Module_10 IS
END Module_10;

ARCHITECTURE behavior OF Module_10 IS


  COMPONENT Module_9
  PORT(
     datai1 : IN  std_logic_vector(7 downto 0);
     datai2 : IN  std_logic_vector(7 downto 0);
     datai3 : IN  std_logic_vector(7 downto 0);
     datai4 : IN  std_logic_vector(7 downto 0);
     wr1 : IN  std_logic;
     wr2 : IN  std_logic;
     wr3 : IN  std_logic;
     wr4 : IN  std_logic;
                    rd1 : in  STD_LOGIC;
     rd2 : in  STD_LOGIC;
     rd3 : in  STD_LOGIC;
     rd4 : in  STD_LOGIC;
     wclock : IN  std_logic;
     rclock : IN  std_logic;
     rst : IN  std_logic;
     datao1 : OUT  std_logic_vector(7 downto 0);
```

```
    datao2 : OUT  std_logic_vector(7 downto 0);
    datao3 : OUT  std_logic_vector(7 downto 0);
    datao4 : OUT  std_logic_vector(7 downto 0)
   );
END COMPONENT;
For dut : Module_9 use entity work.Module_9(Behave);

--Inputs
signal datai1 : std_logic_vector(7 downto 0) ;
signal datai2 : std_logic_vector(7 downto 0) ;
signal datai3 : std_logic_vector(7 downto 0) ;
signal datai4 : std_logic_vector(7 downto 0) ;
signal wr1 : std_logic ;
signal wr2 : std_logic ;
signal wr3 : std_logic ;
signal wr4 : std_logic ;
     signal rd1 : std_logic ;
signal rd2 : std_logic ;
signal rd3 : std_logic ;
signal rd4 : std_logic ;
signal wclock : std_logic ;
signal rclock : std_logic ;
signal rst : std_logic ;

     --Outputs
signal datao1 : std_logic_vector(7 downto 0);
signal datao2 : std_logic_vector(7 downto 0);
signal datao3 : std_logic_vector(7 downto 0);
signal datao4 : std_logic_vector(7 downto 0);
```

```
BEGIN
dut: Module_9 port map
(datai1,datai2,datai3,datai4,wr1,wr2,wr3,wr4,rd1,rd2,rd3,rd
4,wclock,rclock,rst,datao1,datao2,datao3,datao4);

        p1 :process is
        begin
                wclock <= '0' , '1' after 10ns;
                wait for 20ns;
        end process;


        p2 :process is
        begin
                rclock <= '0' , '1' after 10ns;
                wait for 20ns;
        end process;


        test: process is
        begin


        --reset check 1st time
        rst <= '1';
        datai1 <= "00000001";
        datai2 <= "00000011";
        datai3 <= "00000111";
```

```
        datai4 <= "00001111";
        wr1<='1';
        wr2<='1';
        wr3<='1';
        wr4<='1';

        rd1<='0';
        rd2<='0';
        rd3<='0';
        rd4<='0';


        assert (datao1<="00000000") and
(datao2<="00000000") and (datao3<="00000000") and
(datao4<="00000000")
        report "reset error"
        severity error;


        --writing in all datai(1,2,3,4) 1st time
        wait for 10 ns;
        rst <= '0';
        datai1 <= "11111111";
        datai2 <= "10000000";
        datai3 <= "11110000";
        datai4 <= "11001100";
        wr1<='1';
        wr2<='1';
        wr3<='1';
        wr4<='1';
```

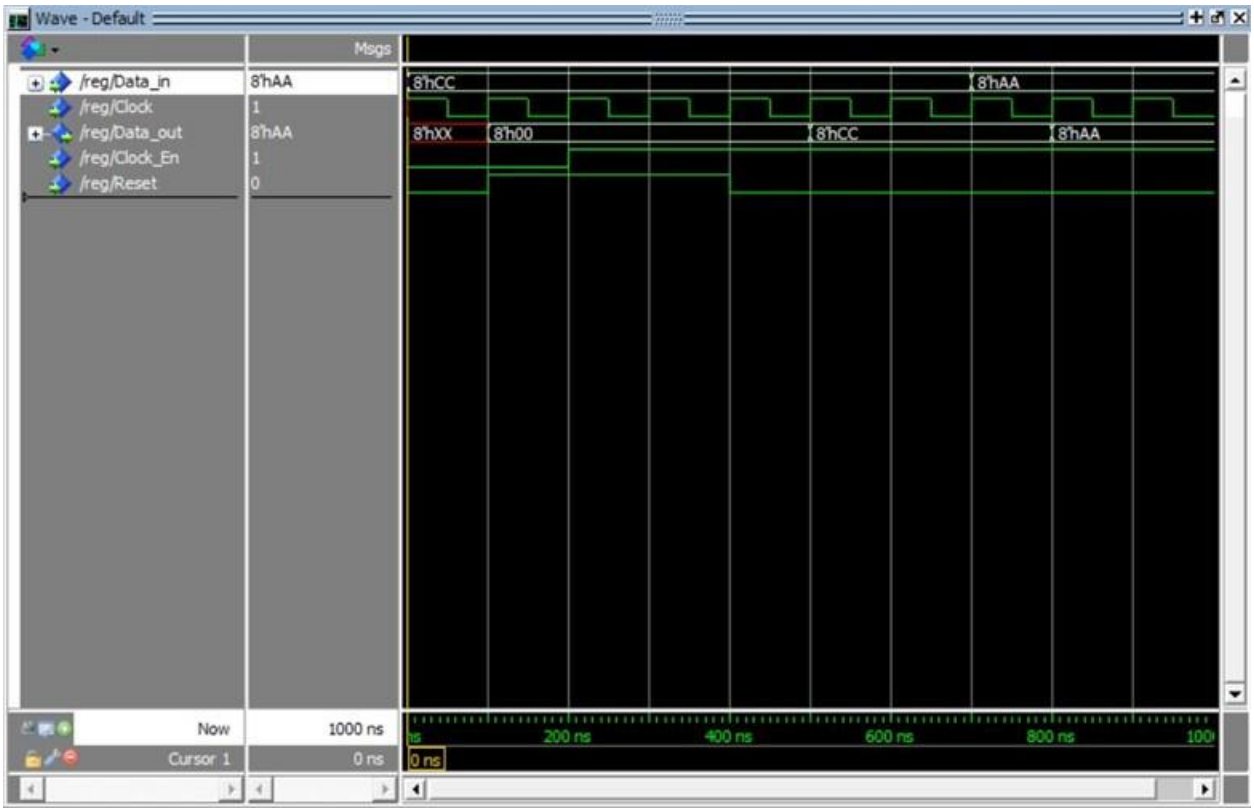
```
        rd1<='0';
        rd2<='0';
        rd3<='0';
        rd4<='0';


        assert (datao1<="00000000") and
(datao2<="00000000") and (datao3<="00000000") and
(datao4<="00000000")
        report "writing error"
        severity error;



        --writing in all datai(1,2,3,4) 2nd time
        wait for 20 ns;
        rst <= '0';
        datai1 <= "11111110";
        datai2 <= "11000000";
        datai3 <= "00000011";
        datai4 <= "11111100";
        wr1<='1';
        wr2<='1';
        wr3<='1';
        wr4<='1';

        rd1<='0';
        rd2<='0';
        rd3<='0';
```

```
        rd4<='0';


        assert (datao1<="00000000") and
(datao2<="00000000") and (datao3<="00000000") and
(datao4<="00000000")
        report "Writing error"
        severity error;



        --reading what was written in 1st time
        wait for 20 ns;
        rst <= '0';
        datai1 <= "11111111";
        datai2 <= "10000000";
        datai3 <= "11110000";
        datai4 <= "11001100";
        wr1<='0';
        wr2<='0';
        wr3<='0';
        wr4<='0';

        rd1<='1';
        rd2<='1';
        rd3<='1';
        rd4<='1';
```

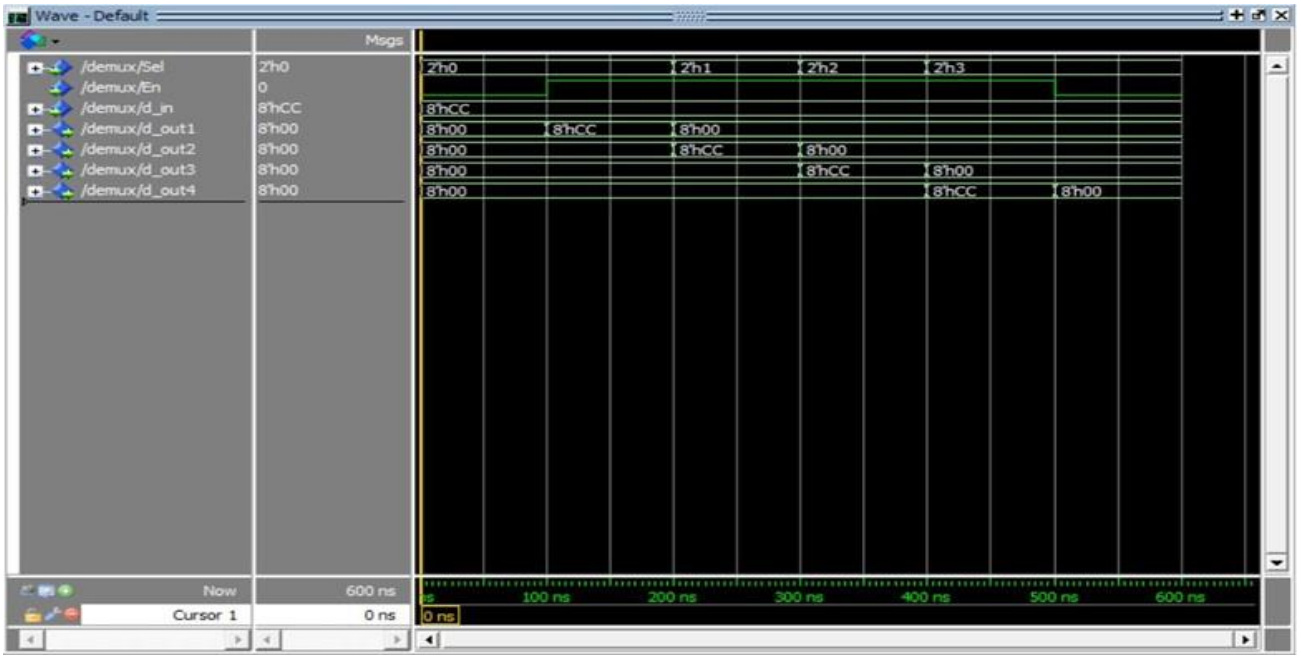
```
        assert (datao1<="11111111") and
(datao2<="10000000") and (datao3<="11110000") and
(datao4<="11001100")
        report "Reading error"
        severity error;


        --reset check 2nd time
        wait for 20 ns;
        rst <= '1';
        datai1 <= "00000001";
        datai2 <= "00000011";
        datai3 <= "00000111";
        datai4 <= "00001111";
        wr1<='1';
        wr2<='1';
        wr3<='1';
        wr4<='1';
        rd1<='0';
        rd2<='0';
        rd3<='0';
        rd4<='0';

        assert (datao1<="00000000") and
(datao2<="00000000") and (datao3<="00000000") and
(datao4<="00000000")
        report "reset error"
        severity error;
        end process;
END;
```

Appendix C

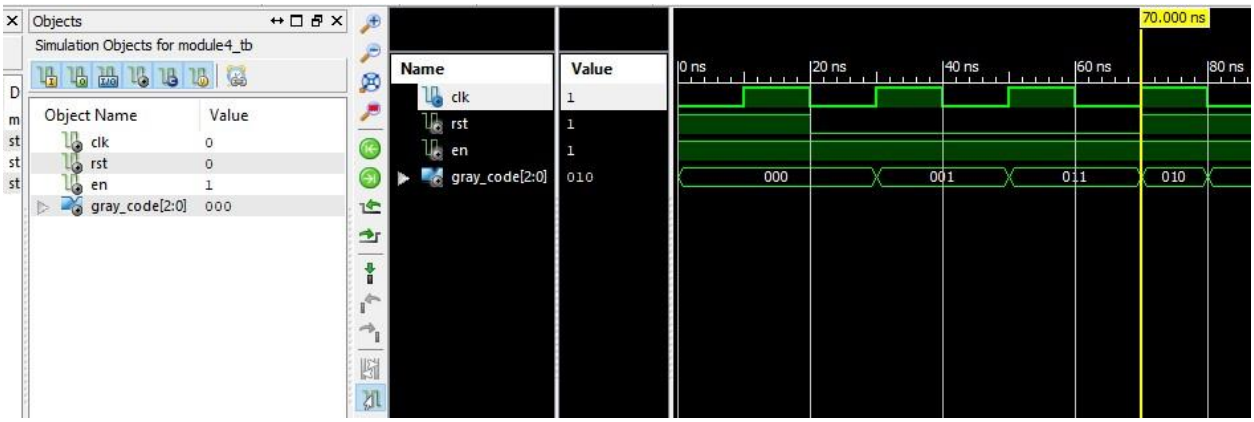
- **Module 1**



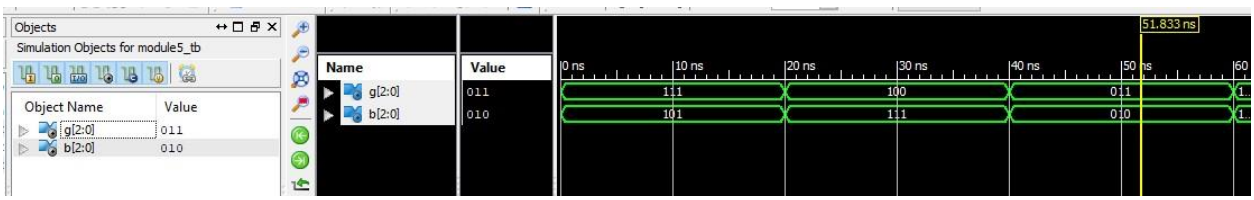
- **Module 2**



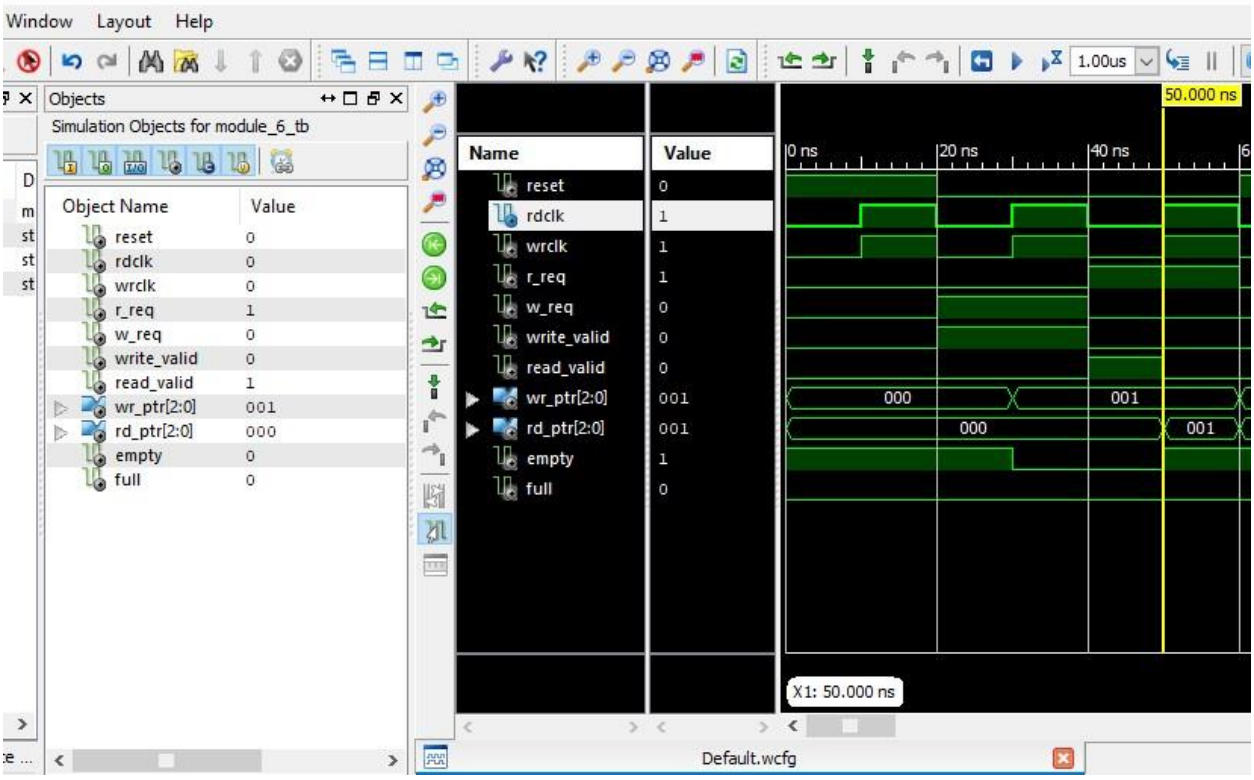
- **Module 4**



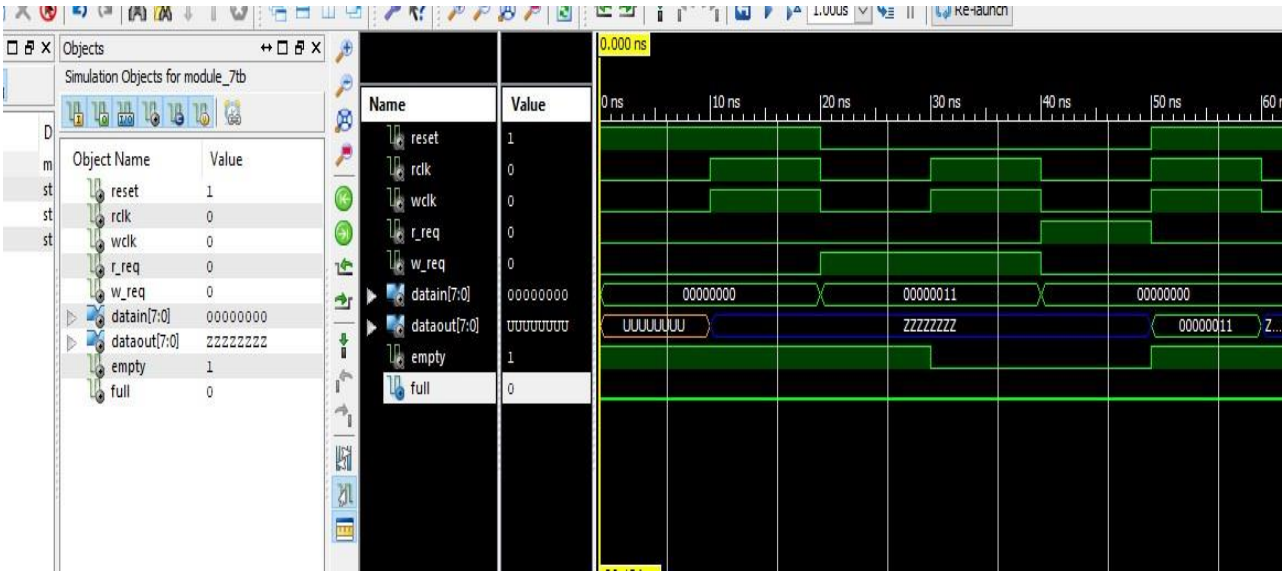
- **Module 5**



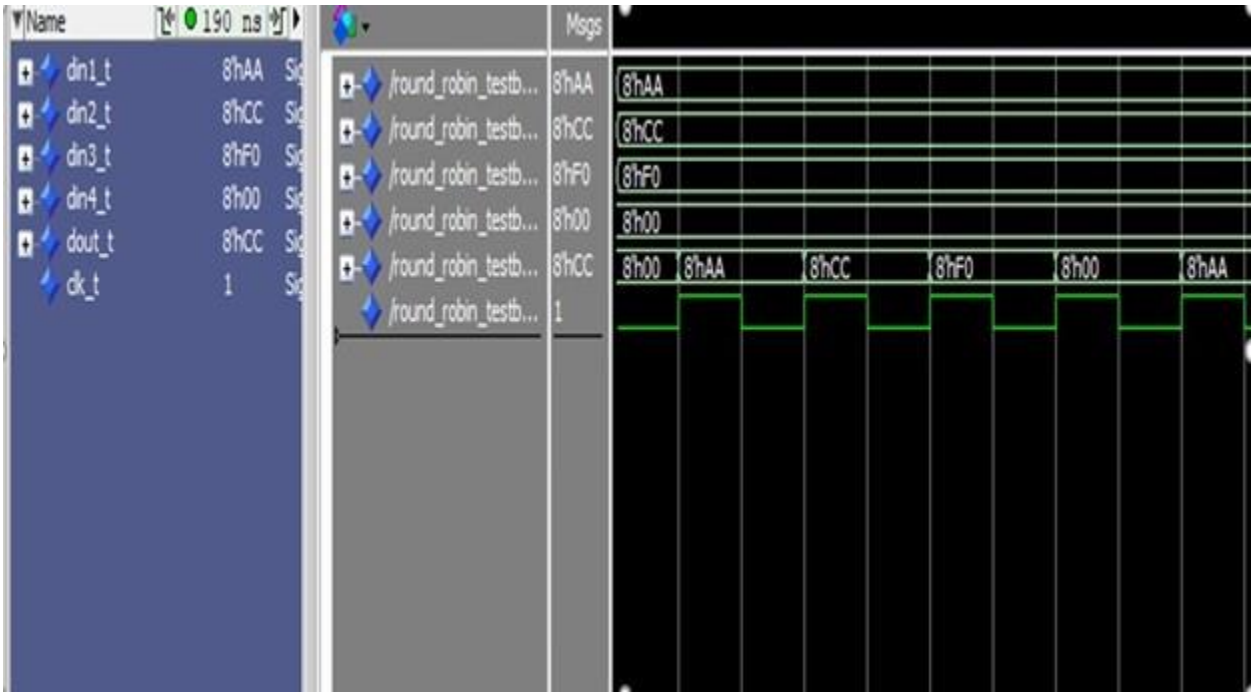
- **Module 6**



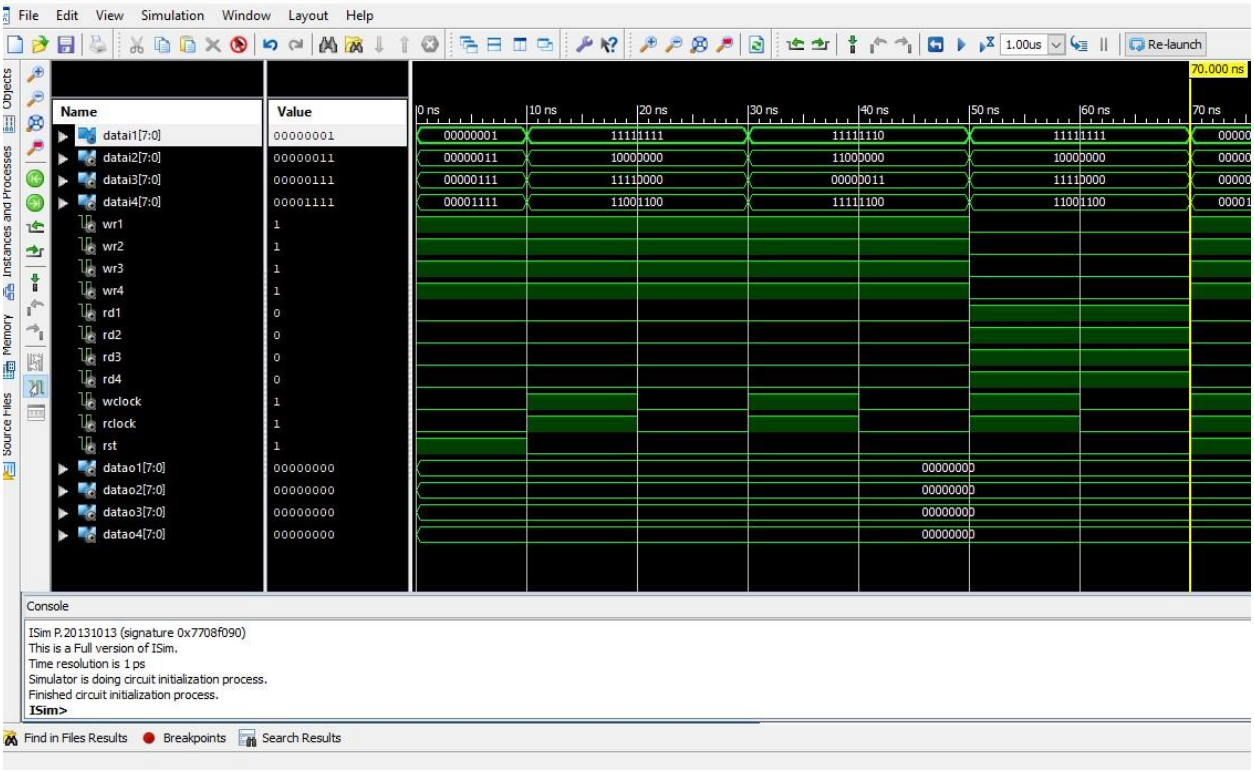
- **Module 7**



- **Module 8**


Name
190 ns
din1_t
8'hAA
din2_t
8'hCC
din3_t
8'hF0
din4_t
8'h00
dout_t
8'hCC
clk_t
1
Msgs
/round_robin_testb...
8'hAA
8'hCC
8'hF0
8'h00
8'hCC
1
8'h00
8'hAA
8'hCC
8'hF0
8'h00
8'hAA

- **Module 10**


File Edit View Simulation Window Layout Help
Re-launch
1.00us
70.000 ns
Name
Value
datai1[7:0]
datai2[7:0]
datai3[7:0]
datai4[7:0]
wr1
wr2
wr3
wr4
rd1
rd2
rd3
rd4
wclock
rclock
rst
datao1[7:0]
datao2[7:0]
datao3[7:0]
datao4[7:0]
0 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
Console
ISim P.20131013 (signature 0x7708f090)
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
Find in Files Results
Breakpoints
Search Results

Task Distribution List

NAME	ID	Task
Ahmed Khaled saad ali Mekheimer	1809799	Module6,7,9,10 Design implementation Test & simulation results
Mohammed Makram mahrous	19p2645	Design Flow Conclusion
Omar Yehia abdelfattah	18p7177	Module 1,2,8 Design implementation Schedular design and FSM implementation
Mohammed Mostafa abdalla	18p9474	Module 3,4,5 Design Implementation
Youssef fawzy ali	16p3087	Introduction Literature review conclusion